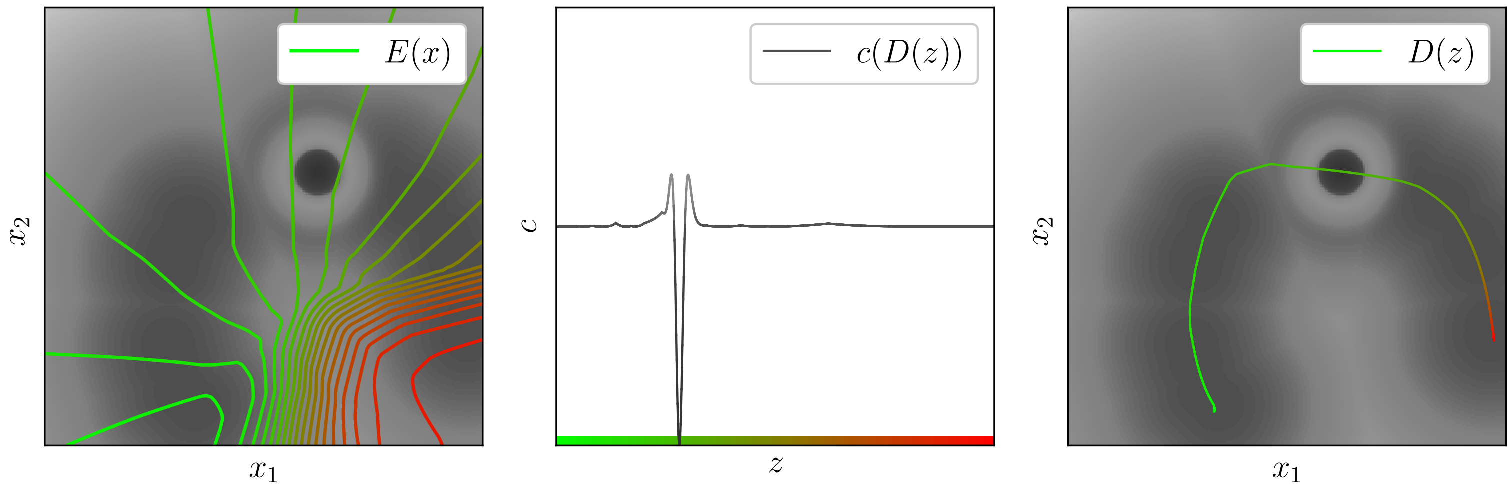
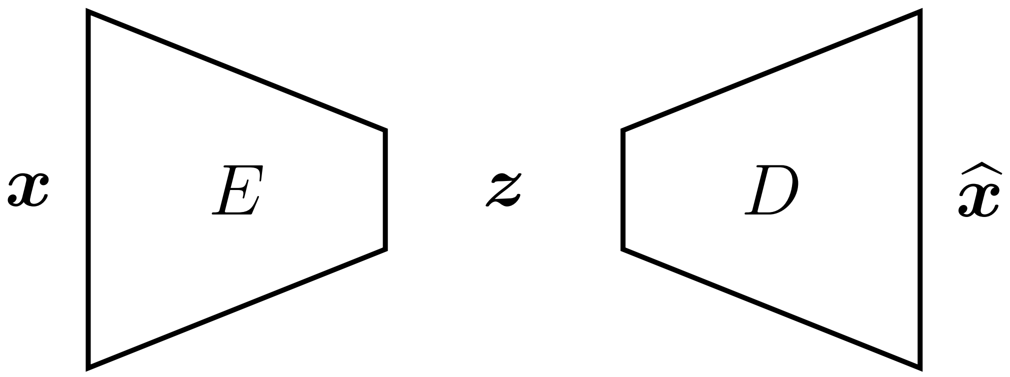
Department of Precision and Microsystems Engineering

Autoencoder enabled global optimization

Julian Schumann

Report no	: EM-2021.063
Coach	: Alejandro M. Aragón
Professor	: Alejandro M. Aragón
Specialization	: Engineering Mechanics
Type of report	: Master Thesis
Date	: 14/08/2021





Precision and Microsystems Engineering

Challenge the future

Autoencoder Enabled Global Optimization

Julian F. Schumann, Alejandro M. Aragón

August 2021

1 Autoencoder Enabled Global Optimization

Abstract

High-dimensional optimization problems with expensive and non-convex cost functions pose a significant challenge, as the non-convexity limits the viability of local optimization, where the results are sensitive to initial guesses and often only represent local minima. But as the number of expensive cost function evaluations required for a full exploration of the search space grows exponentially with the increasing number of dimensions, the use of standard global optimization algorithms is also not practical. To overcome this obstacle and to lower the dimensionality of the problem, the use of an autoencoder for model order reduction is proposed. In the resulting lower dimensional space, standard global optimization methods can then be utilized, as fewer cost functions evaluations are necessary. For problems with comparatively more expensive cost functions, this optimization includes the employment of a surrogate model, which reduces the necessary number of these computationally expensive evaluations further. This proposed method is then tested firstly on a number of benchmark functions, where it shows the ability to find global optima under certain conditions. Secondly, the proposed method is used to solve a compliance minimization problem, where it shows the ability to improve upon a large number of designs generated by local optimization.

1.1 Introduction

Ever since Newton trying to minimize the resistance of a radial symmetric body in a fluid flow [1], countless optimization problems have occupied the minds of scholars and researchers. Today, as humanity has to reconcile the desire of an increasing population for improved living conditions with a finite amount of resources [2], the importance of optimization only grows, because it enables greater efficiency in the use of materials and energy [3, 4, 5, 6]. Thanks to an increase in computational power over the years [7], ever more complex problems in numerous field such as engineering, medicine or economics have become solvable [8, 9, 10]. But high-dimensional problem with a non-convex cost function, where the global optimum to be found is one of many local optima, still pose a challenge. This is caused by the so called "curse of dimensionality", where the number of cost function evaluations required for a thorough investigation of the search domain can increase exponentially with the number of dimensions [11, 12]. This becomes even more troublesome for more expensive problems, since more computationally intensive cost functions reduce the feasible number of cost function evaluations, and therefore limit the size and consequently dimensionality of the space that can be explored. Consequently, for many such expensive and high-dimensional problems, like they are for example encountered in the field of topology optimization, gradient based local optimization algorithms are used [13, 14, 15]. Yet unless the initial guess was extraordinarily lucky, such methods will get caught in local minima, even in comparatively well-behaved problems like compliance minimization [16, 17, 18].

When trying to overcome the limitations of local optimization and improve upon its results, one can usually turn towards two types of global optimization algorithms, namely surrogate model-based Bayesian optimization and global random search [19]. Bayesian optimization is designed for problems with expensive cost function evaluations, but struggles with a high dimensionality, with even recent works only considering problems with a maximum of a few hundred variables [20, 21]. The main reason for this is the construction of the surrogate model, which requires the repeated inversion of a large (size is a multiple of the number of dimensions), dense and often ill-conditioned (near singular) matrix [19], which is not only time intensive, but can also lead to memory issues [22]. On the other hand, global random search algorithms can be used for higher dimensional problems, but are reliant on a relatively cheap cost function, due to the higher number of required cost function evaluations. The most common type here is population-based search like differential evolution [23] and particle swarm optimization [24].

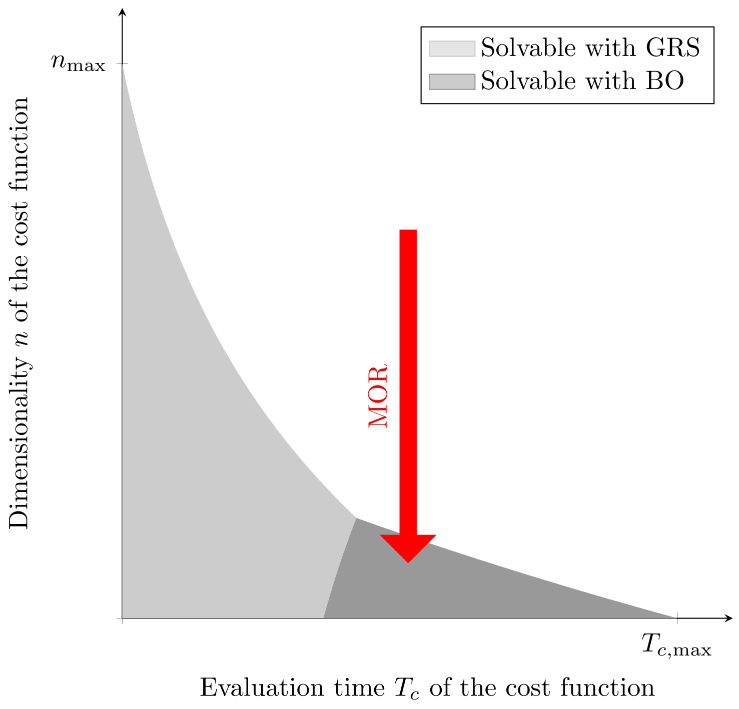
Many researchers have tried to construct better algorithms for high-dimensional problems, but they are all limited due to the so-called “no free lunch theorem”, according to which every optimization algorithm will perform the same, when averaged over all possible optimization problems [25, 26]. Consequently, a certain algorithm can only be more efficient than others on a certain set of cost functions. In some cases, this is acknowledged, and for example only cost functions with non-deceptive gradients, meaning that following the slope of the cost function can lead to the global optimum, are considered [27]. In other cases, this is ignored, and the algorithm is improved by tuning some hyperparameters of already existing algorithms [8, 28]. But as the efficiency is usually only tested on benchmark functions unrelated to real-world problems, the utility of such algorithms for instance in regard to real engineering problems can be expected to be limited [27, 19].

In this work, the assumption is made that the domain of the cost function, in which most or all local minima and therefore most of the better cost function values lie, has a intrinsically lower dimensionality then the original problem. Based on this, model order reduction can then be used to limit the search space to this domain. For this purpose, autoencoders - a form of neural network - are chosen, as they outperform common methods like proper orthogonal decomposition, especially if the searched domain, which in the context of autoencoders is also called latent space, has nonlinear features [29, 30, 31, 32, 33, 34]. Global optimization only over the latent space with a comparatively low dimensionality then becomes possible. As global optimization over such a low-dimensional space is covered extensively in the literature, be it global random search for cheaper cost functions [35, 36, 37, 38] or Bayesian optimization for more expensive cost functions [39, 40], the main focus of this work will be on the use of the autoencoder.

To the best of the authors’ knowledge, the concept of using autoencoders to allow for global optimization in latent space has been first proposed by Costa in 2008 [41], although it is applied to a relatively low-dimensional problem (120 dimensions, constrained wind-hydro coordination problem) with a reduction in dimensionality by only a factor of 2. Similar work has also been pursued also by Miranda et al. [42]. A larger reduction in dimensionality can be found in the work of Gao et al. [43, 44], where the dimensionality of the problem of fitting a geology model to data could be reduced from 38400 dimensions down to 140. Further additions have been made by Eismann et al. [45], where the autoencoder is extended with a neural network surrogate model mapping the 20-dimensional latent space representation to its original cost function value, to enforce distance between good and bad designs in the reduced space. This method leads to improved designs for a 9408-dimensional drag minimization problem, with similar benefits in the data structure in latent space being suggested by Wang et al. [46]. Jiang et al. [47] found that a higher fitness of the training samples used to train the autoencoder can improve the fitness of designs generated from decoding the latent space, showing the value of using local optimization on the training samples before fitting autoencoder. Additionally, the fitness of these designs can be improved further by using post-processing techniques, by using local optimization on the decoded samples in the high-dimensional search space. Another idea has been suggested by Kudyshev et al. [48], proposing the use of an adversarial autoencoder to enforce a certain distribution of designs in the reduced space, when trying to optimize phonetic metamaterials. Autoencoders with the purpose of reducing the search space have been employed in other works as well, for example in the design of electromagnetic circuits [49, 50], optical microstructures [51] or mechanical structures such as springs [52] or wheels [53] and even the design of molecules in chemistry [54].

While all these works include autoencoders to reduce the dimensionality of optimization problems, they only consider their use in the environment of the specific problem contemplated. Consequently, there is a lack of justification, besides the results of the optimization, for the employment of this method, and it is therefore unclear for what kind of other optimization problems this approach could be used. To address this, the authors propose a generalized method for autoencoder enabled global optimization (see section 1.2), and through the optimization of constructed benchmark functions, the conditions for a successful application are explored (see section 1.3). Furthermore, the method will be used to optimize a real-world compliance minimization problem, which this approach to the best of the authors' knowledge has previously not been used for (see section 1.4). The autoencoder setup and encoding method, a signed distance field, are also novel for this application (see appendix I.2).

1.2 Methods



Solvable with GRS
Solvable with BO
$n_{\max}$
Dimensionality n of the cost function
MOR
$T_{c,\max}$
Evaluation time T_c of the cost function

Figure 1: Visualization of different optimization problems, which can possibly be solved either with global random search (GRS) or Bayesian optimization (BO). Model order reduction (MOR) reduces the dimensionality of the problem and can make optimization of previously unsolvable problems possible. It has to be noted that this figure is not to scale, and that its shape may vary depending on the problems' parameters.

There are two main factors that determine if, and possibly how, an optimization problem

$$\boldsymbol{x}_{\min} = \underset{\boldsymbol{x} \in X}{\operatorname{argmin}}\, c(\boldsymbol{x}), \quad X \subset \mathbb{R}^n \tag{1}$$

can be solved with generalized methods in a limited amount of time. These are the dimensionality n of the problem and the time T_c it takes to evaluate one instance of the cost function $c(\boldsymbol{x})$, with both values having upper limits ($n_{\max}$ and $T_{c,\max}$ respectively, see Figure 1). For example, if T_c is larger than the allowed time, no optimization is possible. And even for $T_c = 0$, the time it takes for even the simplest global random search algorithm to come up with new sample points limits the number of cost function evaluations and therefore the dimensionality of the space that can be explored sufficiently.

It is rather unlikely that one could reduce T_c without changing the behavior of the cost function as well as its optimum and consequently the final result of the optimization. Therefore, the only possibility to optimize previously unsolvable problems would be to only explore parts of the search space X, as this requires fewer cost function evaluations. In this work, the authors then assume that most, if not all, of the local minima and also the global optimum of the cost functions $c(\boldsymbol{x})$ to be optimized lie on a m-dimensional manifold $X' \subset \mathbb{R}^m$ inside the search space (for further discussion see Appendix B).

Consequently, one would then only need to optimize over this reduced domain $X' \subset X$, and as $m < n$, this would be possible in a feasible amount of time. To optimize such problems, the following four step method (see Figure 2 for a symbolic representation) is then proposed, where model order reduction, using an autoencoder, is employed to approximate the domain X' based on some training samples.

1. A training set for the autoencoder is generated, which should be representative of the manifold X'. This is done by randomly sampling N points $\boldsymbol{x}_i \in \mathbb{R}^n$ in the design domain X. The cost $c(\boldsymbol{x}_i)$ of these samples are then improved by using local optimization to advance them in the direction of their corresponding local minima of c. Here, $\boldsymbol{X}_\lambda$ denotes the set of samples after the λ-th step of local optimization (LO is the corresponding operator, which can stand for any arbitrary local optimization algorithm):
$$\boldsymbol{X}_\lambda = LO\left(\boldsymbol{X}_{\lambda-1}\right) = LO^\lambda\left(\boldsymbol{X}_0\right) \tag{2}$$
The value of λ should be large enough so that the resulting training samples $\boldsymbol{X}_\lambda$ best possibly represent the reduced space X', with the use of local optimization being the only reliable way to find samples $\boldsymbol{x} \in X'$ (see Appendix C for an example in compliance minimization). But one should also aim to not make λ too big, as this would lead to unnecessarily long computation times.
To generate these samples, one could use methods like Adam [55] (see Appendix A.1) or alternatively the SIMP method [56] in the case of topology optimization problems like compliance minimization (see Appendix A.2). While both these methods are gradient based and therefore require a differentiable cost function $c(\boldsymbol{x}) \in C^1$, this is not necessarily a requirement, as the use of zeroth order local optimization like the Nelder-Mead algorithm [57] is also possible. To avoid the problem of multiple different starting points resulting in similar designs after local optimization, sampling techniques like deflation [58] can be used. In this method, new samples are guided away from already explored samples by modifying the cost function.

2. In a second step, the samples in $\boldsymbol{X}_\lambda$ (or a specific subset of $\boldsymbol{X}_\lambda$) are used to train an autoencoder network (see Appendix A.6). While there are multiple different methods that can be used here (autoencoder, variational autoencoder (VAE) [59], generative adversarial network (GAN) [60], adversarial autoencoder (AAE)[61], autoencoder with surrogate model network [45]), this step will always produce a decoder network D, which allows the transformation of a latent space representations $\boldsymbol{z}_i \in Z \subset \mathbb{R}^m$ into the design space X. But the decoded samples $D(\boldsymbol{z}_i) \in \mathbb{R}^n$ will only fill a m-dimensional manifold X_Z - also referred to as decoded latent space in this work - of the n-dimensional design space ($D : Z \to X_Z \subset X$). For example, if $n = 2$ and $m = 1$, X_Z will be a continuous and possibly curved line in the two-dimensional design space X (see Appendix D). The goal of this step is to approximate X' with X_Z as best as possible, so that hopefully the global optimum is included in X_Z.

3. A global optimization approach is used to optimize the design in latent space by minimizing the cost function $c_\mu(\boldsymbol{z})$, where $\mu > 0$ might be used to account for cases where the global optimum does not lie in X_Z itself, but still close to it:
$$c_\mu(\boldsymbol{z}) = c\left(\widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}\right)\right), \quad \widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}\right) = LO^\mu\left(D(\boldsymbol{z})\right) \tag{3}$$
Depending on the time T_c necessary to evaluate this cost function, one can apply global random search methods like differential evolution [23] (see Appendix A.3) for cheaper cost functions or Bayesian optimization (see Appendix A.4) for more expensive ones. If the time required for both methods is similar, it is often better to choose global random search methods, as they generally tend to produce superior results (see Appendix K.3).
Here it should be mentioned that for $\mu > 0$, the cost function itself might become discontinuous, so gradient based methods are likely not feasible at this step.

4. A local optimum $\boldsymbol{z}^*$ in latent space does not necessarily correspond to a local optimum in the design space:
$$\left.\frac{\partial c\left(\widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}\right)\right)}{\partial \boldsymbol{z}}\right|_{\boldsymbol{z}=\boldsymbol{z}^*} = \left.\frac{\partial c\left(\boldsymbol{x}\right)}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}^*\right)} \left.\frac{\partial \widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}\right)}{\partial \boldsymbol{z}}\right|_{\boldsymbol{z}=\boldsymbol{z}^*} = \boldsymbol{0} \not\Rightarrow \left.\frac{\partial c\left(\boldsymbol{x}\right)}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}=\widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}^*\right)} = \boldsymbol{0} \tag{4}$$
Consequently, the design can be improved by advancing the design $\widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}^*\right)$ further towards the final design $\boldsymbol{x}^*$ by ν steps. This post-processing step will result in the final design:
$$\boldsymbol{x}^* = LO^\nu\left(\widehat{\boldsymbol{x}}_\mu\left(\boldsymbol{z}^*\right)\right) \tag{5}$$

If different forms of local optimization are used for global optimization and post-processing, the following alternative is also possible:

$$\boldsymbol{x}^* = LO^{\nu}\left(D\left(\boldsymbol{z}^*\right)\right) \tag{6}$$

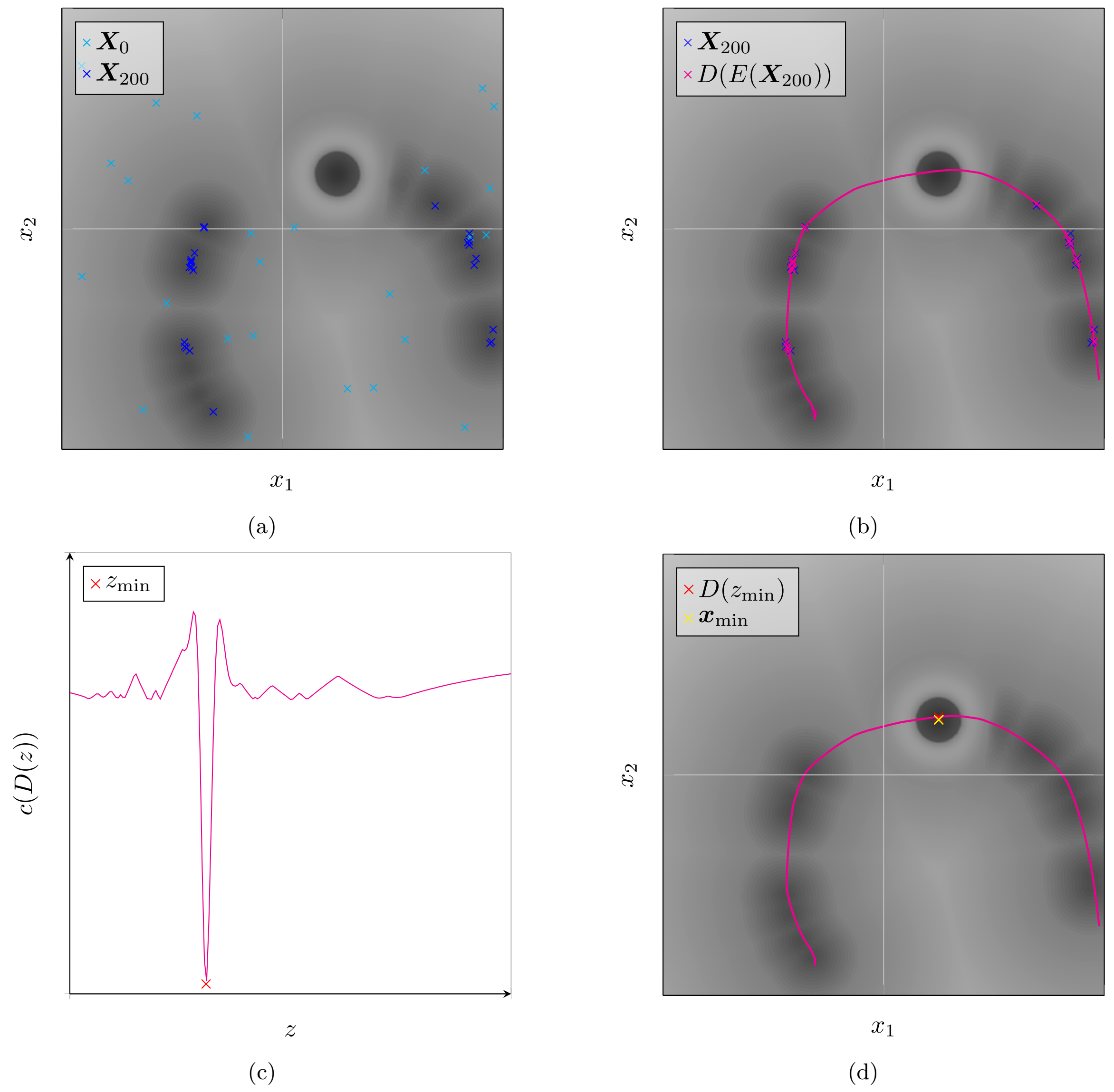


Figure 2: A symbolic representation of the four steps of the proposed method for a case of $n = 2$ and $m = 1$. In the first step (2a), a training set $\boldsymbol{X}_{200}$ is created by 200 iterations of gradient based local optimization, starting from random samples $\boldsymbol{X}_0$. An autoencoder is trained on this training set, leading to the decoded latent space X_Z (the magenta line in 2b and 2d). Over this manifold, global optimization is performed to find z_{min} (2c), from which on the global minimum $\boldsymbol{x}_{\text{min}}$ is found using post processing (2d). In the subfigures 2a, 2b, and 2d, the gray value are being used to represent the cost function $c(\boldsymbol{x})$, with lighter colors representing higher values.

1.3 Benchmark functions and proof of concept

The proposed method is tested on a number of different optimization problems, with the first cost function c_1 being specifically designed so that it fulfills the assumption made about cost functions on

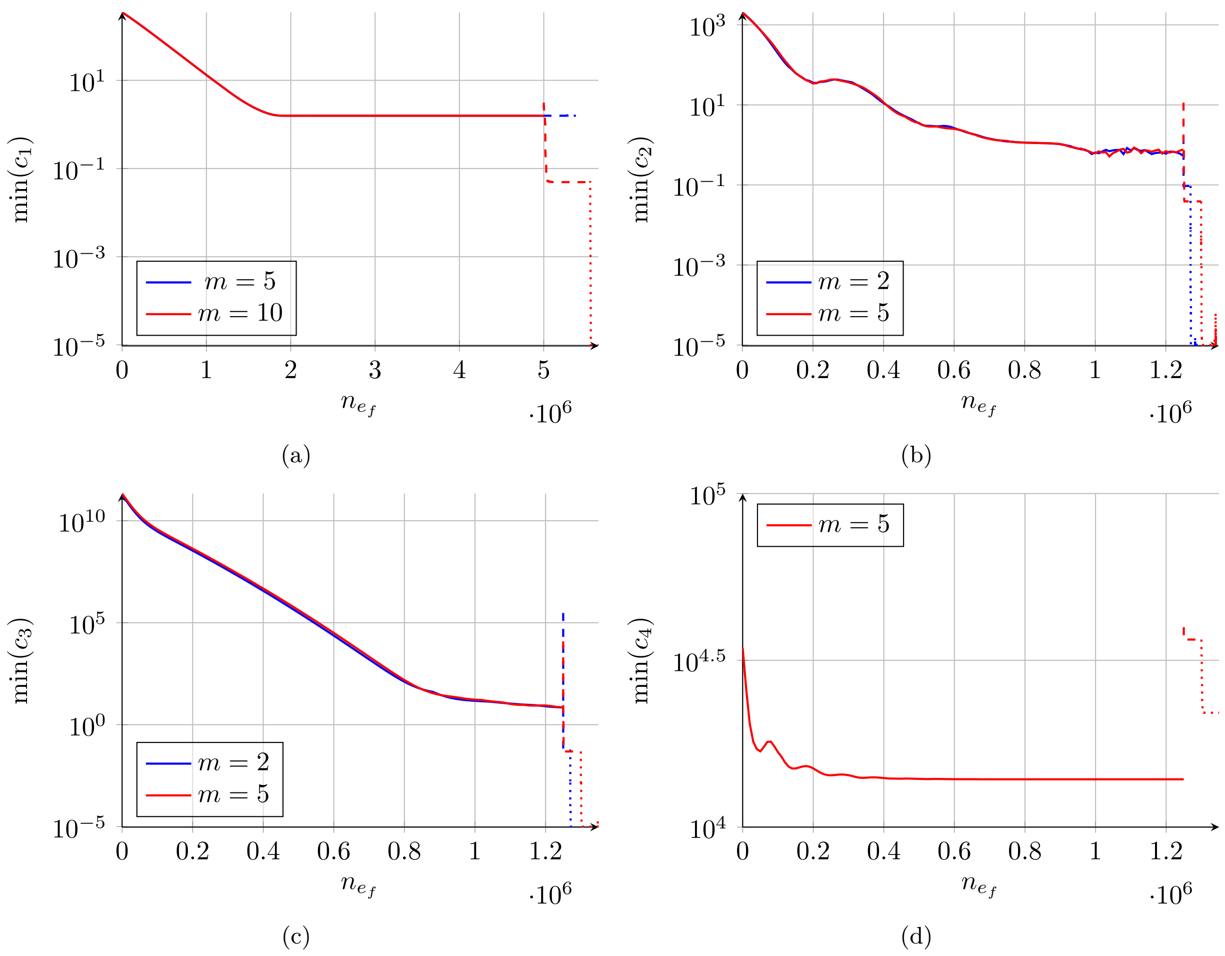


Figure 3: The results of optimizing the functions c_1, c_2, c_3 and c_4. The continuous lines describe the first step, where λ steps of local optimization are used on 10000 random initial points. The dashed line shows the minimum during the optimization over latent space, while the dotted line shows the best compliance during post-processing. It has to be noted that the number of function evaluations n_{e_f} is a discrete number, but due to the high number of available data points, and to better distinguish between different steps of the process with different types of lines, continuous lines have been used.

which the proposed method is based (see Appendix B).

$$c_1(\boldsymbol{x}) = \min_i ||\boldsymbol{x} - \boldsymbol{x}_i||^2 + \left(1 - \exp\left(-10 \left\|\boldsymbol{x} - \boldsymbol{x}_1\right\|^2\right)\right)\left(\frac{2}{5} + \exp\left(-10 \left\|\boldsymbol{x} - \boldsymbol{x}_1\right\|^2\right)\right) \tag{7}$$

Here, $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{2500}\}$ are 2500 points arrayed on a $M = 10$ dimensional subdomain in the $n = 1000$ dimensional design space $X_1 = [-1, 1]^n$ (the exact creation of $\boldsymbol{X}$ can be seen in equation (67) and the precise optimization process used in Appendix F). Three more problems are benchmark functions taken from the work of Abualigah et al. [62]:

$$
\begin{aligned}
c_2(\boldsymbol{x}) =& 1 + \frac{1}{4000}\sum_{i=1}^{n}\left(x_i^2\right) - \prod_{i=1}^{n}\left(\cos\left(\frac{x_i}{\sqrt{i}}\right)\right) \\
c_3(\boldsymbol{x}) =& \frac{\pi}{n}\left(10\sin\left(\pi y(x_1)\right)^2 + \sum_{i=1}^{n-1}\left(y(x_i)-1\right)^2\left(1+10\sin\left(\pi y(x_{i+1})\right)^2 + u(\boldsymbol{x})\right)\right) \\
& y(x) = \frac{x+5}{4}, \quad u(\boldsymbol{x}) = \sum_{i=1}^{n} 100\max\left\{0, |x_i| - 10\right\}^4 \\
c_4(\boldsymbol{x}) =& \sum_{i=1}^{n}\left(-x_i\sin\left(\sqrt{|x_i|}\right)\right) + 418.9829n
\end{aligned}
\tag{8}
$$

All of these three functions are optimized over an $n = 100$ dimensional domain, with $X_2 = X_4 = [-500, 500]^n$ and $X_3 = [-50, 50]^n$. The exact implementation of the optimization process can be seen in Appendix G. All four functions in the section have a global minimum of $c_i = 0$.

In Figure 3 it can be seen that the global optimum of functions c_2 and c_3 can be found no matter the latent space dimensionality m. Meanwhile, the global optimum of c_1 can only be found for $m = 10$, and the global optimum of c_4 cannot be found at all.

The results for c_1 are as expected, as the function was specifically designed so it could be optimized by the proposed method if X_Z could approximate X'. This was indeed possible for $M = m = 10$, but impossible for $M > m = 5$. Furthermore, the results of c_4 also are according to expectations, as the cost function has local minima distributed more or less regularly throughout the whole domain X_4. Therefore, the dimensionality of X' would also be n, and consequently, the proposed method failed. In contrast to that stand the results for c_2 and c_3, where the local optima are also evenly spread over X_2 and X_3 respectively. But the main difference here is that the global optimum of these function is precisely in the middle of the cluster of local minima, and when encoding an evenly spread higher dimensional manifold, autoencoders tend to include the mean of such clusters (see Appendix E). But as cost functions with such peculiar behavior seem unlikely to occur outside of constructed benchmark problems, this possibility for successful optimization can likely be discarded for real world problems. A more detailed discussion and additional results can be found in Appendices F and G.

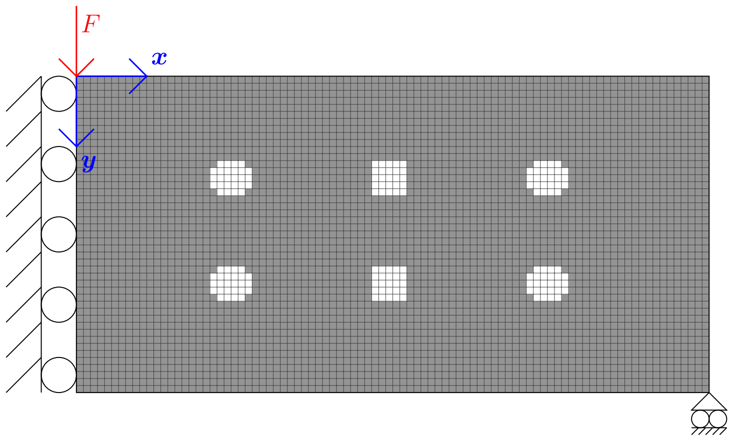
1.4 Compliance minimization

After testing the proposed method on some constructed benchmark functions, it lastly is applied towards a real-world problem. This will be a topology optimization problem, the minimization of the compliance C of a Messerschmitt-Bölkow-Blohm (MBB) beam with enforced void elements (see Figure 4a):

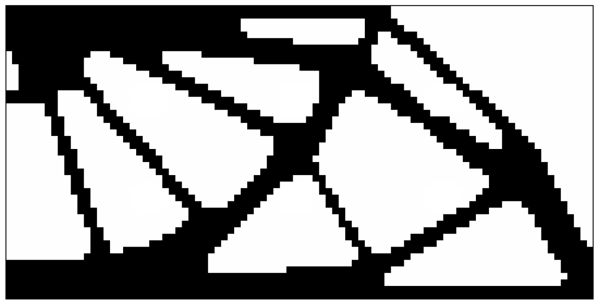
$$
\begin{aligned}
\min_{\boldsymbol{x}\in[0,1]^{n_y\times n_x}} & C(\boldsymbol{x}) = \langle \boldsymbol{u}(\boldsymbol{x}), \boldsymbol{F}\rangle \\
\boldsymbol{K}(\boldsymbol{x})\boldsymbol{u}(\boldsymbol{x}) &= \boldsymbol{F} \\
\overline{\boldsymbol{x}} &\overset{!}{=} \rho_0 = 0.4 \\
[x]_{i,j} &\overset{!}{=} 0 \ \ \forall\{i,j\} \in V
\end{aligned}
\tag{9}
$$

Here, $\boldsymbol{K} \in \mathbb{R}^{2n_yn_x\times 2n_yn_x}$ is the stiffness matrix resulting from the material density $\boldsymbol{x}$, and $\boldsymbol{F}$ is the force vector which contains the force F from Figure 4a, while $\boldsymbol{u}$ is the displacement field and V is the set of void elements, whose purpose is to increase the non-convexity of the problem. This would allow for a better assessment of the proposed method in comparison to the standard gradient-based approach that is normally pursued for such problems.

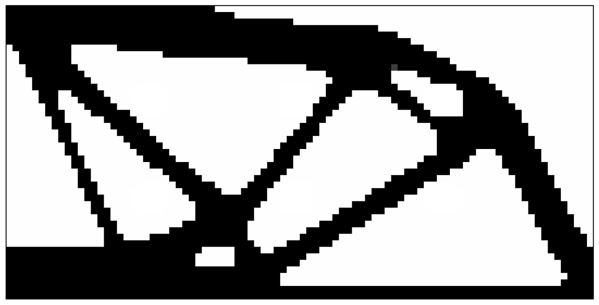
The proposed method is promising for this problem, as it is likely that large parts of this search space are unusable, as intermediate density values ($[x]_{i,j} \notin \{0, 1\}$) are punished (see Appendix A.2). Furthermore, even sparse designs can be undesirable if they include disconnected parts. Consequently, it can be expected that X', the manifold including the global optimum and other stiff designs, is likely far lower dimensional than the original design space. Furthermore, this domain X' will also be highly nonlinear, as most feasible designs will be close to or on the boundary of the search space, which justifies the use of autoencoders for model order reduction employed in the proposed method.



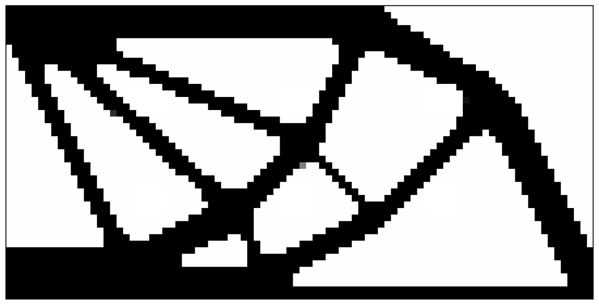
(a) Boundary conditions of the problem.



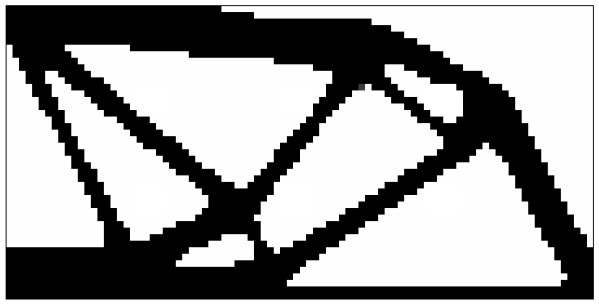
(b) Solution from homogeneous input after 1000 steps. $C \approx 94.86$



(c) Best design from the training set (Appendix I.1). $C \approx 92.65$



(d) Best design after post-processing step using differential evolution for optimization over latent space. $C \approx 91.64$



(e) Best design after post-processing step using Bayesian optimization for optimization over latent space. $C \approx 91.98$

Figure 4: Boundary condition for a MBB beam, and different solutions of this problem. The white elements in 4a are void elements, where the lowest possible stiffness is assumed.

This problem will be solved on a $n_y \times n_x = 45 \times 90$ grid, with a model order reduction to $m = 100$ dimensions. There will be two solutions, with one using a global random search algorithm, namely differential evolution (Appendix A.3) for the optimization over latent space, while the second one will use Bayesian optimization (Appendix A.4). An exact implementation of the proposed method for this compliance minimization problem can be found in Appendix I.

It can be seen in Figure 4 that the designs produced by the proposed method (4d and 4e) have a lower compliance C than both the solution stemming from a homogeneous input using standard SIMP (4b), as well as the sample with the lowest compliance from the training set for the autoencoder (4c), with differential evolution being able to produce the better result than Bayesian optimization. But as the design space is too large to be explored fully using brute force methods (there are around 1.44×10^{810}

sparse solutions for this problem), it cannot be determined if the found solutions are even close to the global optimum.

Table 1: The number of finite element evaluations $n_{e_{FEM}}$ and the time t needed for each step of the proposed method. The first step was parallelized on 50 processor cores, while the second and third step used 100 cores each.

Step	Training data	Training AE	Optimization DE	Optimization BO	Post-processing
$n_{e_{FEM}}$	$1.55 \cdot 10^6$	0	$5.25 \cdot 10^6$	$4.55 \cdot 10^4$	$3.10 \cdot 10^2$
t (in s)	$4.61 \cdot 10^3$	$1.26 \cdot 10^3$	$5.66 \cdot 10^3$	$5.23 \cdot 10^4$	$1.00 \cdot 10^2$

Besides the compliance, a point that has to be considered as well is the computational time and cost. The whole proposed method took at least over three hours to compute with access to 100 processor cores for parallelization. Meanwhile, generating the solution using the standard method with homogeneous input only took five and a half minutes on a single core. As can be seen in Table 1, using differential evolution is faster than Bayesian optimization for the current problem. This is mainly caused by the fact that building the model is expensive compared to evaluating the cost function, on which only 1000 s are actually spent ($T_c \approx 3.5s$ per evaluation, the $M_0 = 1000$ initial evaluations are parallelized on 100 processor cores). But as the computational cost for building the model is not dependent on the cost function expense, it can be extrapolated that for a cost function evaluation time of $T_{c,0} \approx 42s$, Bayesian optimization will become faster (up to a factor of $k_T \approx 5$). Additionally, if parallelization is not possible, this will tilt more favourably towards Bayesian optimization (see Appendix J). On the other hand, these potential time savings come at the potential cost of a reduced reliability in finding the best design possible, as could be seen in this example, where both methods were used to optimize the same cost function, but Bayesian optimization was unable to find a result as good as the one achieved by differential evolution, a global random search method.

But while Bayesian optimization can be used to significantly speed up the optimization over latent space in the case of expensive cost functions, the time needed for the generation of the training data and the training of the neural network are still significant (see Table 1). Two different cases have to be considered here:

- If the cost function is expensive, than the time needed for training the neural network can be neglected. But then the creation of the training set becomes the most expensive part of the proposed method, as in contrary to the optimization over latent space, surrogate models cannot be used, as they would need to span the higher dimensional design space instead of latent space. A possible solution here might be to use the same training set for different problems, but at least according to Appendix K.4, this approach does not look promising.
- If the cost function is cheap, than the training of the neural network becomes the main consumer of computational power. While the number of calculations to train the neural network cannot be changed, one could use superior hardware like GPUs and TPUs [63] to potentially speed up this step of the proposed method.

1.5 Conclusion

Overall, the proposed method can in certain cases find the global optimum or at least superior results compared to other methods for solving optimization problems. However, this comes at a high cost, especially when compared to the simple, gradient based approaches often used for expensive problems like topology optimization. Therefore, its potential usage is most promising for problems where either a wealth of possible problem solutions already exists, rendering the first step of the process unnecessary, or where the cost function is highly non-convex. In such cases, the idea of using an educated guess like the homogeneous distribution as a starting point for gradient based local optimization would likely not lead to a satisfactory result.

Furthermore, it has to be noted that this method relies on the cost function matching the assumptions made. If not, the result produced might be even worse than the training samples used. Consequently, this method cannot be used for every optimization problem, and careful deliberation is necessary to justify its application.

While the proposed method seems promising, there are still points which might deserve further research. Finding a faster method of generating good training samples would improve the viability of this method for more expensive cost functions. Looking closer at the influence of post-processing, and its potential ability to make previously inferior designs in latent space superior in the end could also lead to a better understanding of the method. Additionally, the introduction of methods which allow the estimation of the best size of the latent space before building and training the autoencoder could lead to greater efficiency. And finally, retraining the autoencoder on a set including the optimized results, to then possibly find better results in a second round of optimization could also be explored.

Acknowledgments

The authors would like to thank the Technical University Delft for supporting this work by making available its high-performance computing cluster for their use. Thanks also goes to Professor Fred van Keulen for suggesting the use of signed distance field as a method of encoding.

2 Reflection

2.1 Timeline

I started this master thesis in September 2020. At this time, I did not create a fixed timetable, outlining the planned step of my thesis, as the actual topic of the thesis was not set in stone yet (see below). Consequently, there is no comparison I can make to my planning, and I will leave this point at a brief overview of how I spent my time during the past months (see also Figure 5).

- The first three months of the project where spent mainly on reviewing the literature and preparing for the literature survey. But simultaneously, I also spend time on properly implementing the SIMP algorithm and learning how to work with neural networks.
- After that, I spent December and January mainly on familiarizing myself with working on the high performance computing cluster, and rewriting all my existing code so that it took advantage of the existing computational power. A more detailed reflection on this part can be found in section 2.3.
- The next two months were mainly spent on improving the autoencoders in regard to their ability to encode the types of designs commonly produced as solutions for compliance minimization. This for example includes testing the viability of encoding signed distance fields instead of density fields. I also started writing my report during this time.
- Thereafter, the next two months where spent on implementing the benchmark problems and most of the compliance minimization problems, which can be found in Appendices F, G, and K.
- In June, due to unsatisfactory results from the compliance minimization, I had to repeat some simulations with an updated cost function, leading to the results shown in section 1.4. Furthermore, most parts of the report were written during this month.

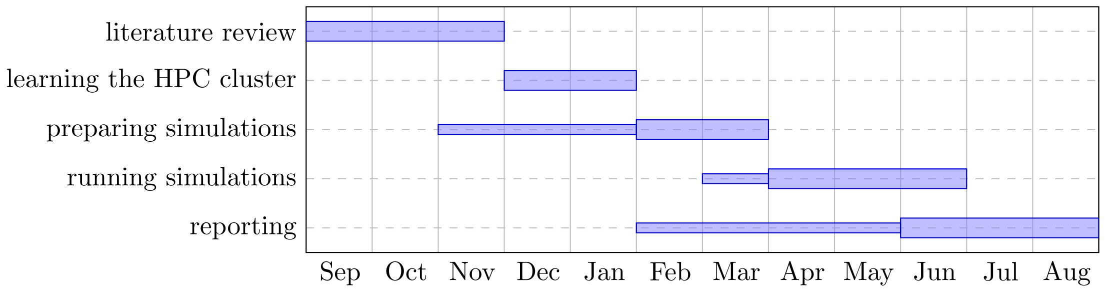


Figure 5: A rough estimation of the time spent on different aspects of my master thesis.

2.2 Change of research question over time

Over the whole project, there was a more or less continuous shift in the goal of the thesis. In the beginning, the specific topic was very open, mainly focusing on the possibility of using autoencoder simply for model order reduction in modelling. But it showed early that research in this direction was already plentiful, with very few possibilities to expand upon this research except finding new models to apply it towards.
Consequently, a shift happened in the research, where I started to focus instead more on topology optimization, the non-convexity of such optimization problems, and the hope that autoencoders might be used to overcome this issue. Based on this, the four main steps of the proposed method were proposed, although this was mainly focused on the application of compliance minimization. Consequently, as an example, most of the simulations in the early part of the thesis work were focused on optimizing the autoencoder simply in regard to minimizing the reconstruction error.
But during the last months of the thesis, the scope of the thesis widened again, now looking at the possibility of using the proposed method more generally for optimization problems. Nonetheless, true

to previous intentions, topology optimization still remained the main application the thesis work considered. Specifically, compliance minimization was selected as the example case due two mainly two reasons. Firstly, the author already had previous experience with implementing this kind of problem, requiring less time to implement the code and speeding up the required calculations for the local optimizer (in this case SIMP was used). Secondly, the cost function for compliance minimization was very cheap compared to other topology optimization problems, as its main cost was the solution of a simple, linear FEM problem. The cheap cost function allowed for faster simulations, and therefore enabled more simulations and consequently the exploration of more variations of the proposed method.
But as was mentioned in the previous sections, the focus on simple compliance minimization problems also lead to unsatisfactory results, problems which were only overcome in the last stretches of the thesis work.

2.3 Implementing parallel processing on the cluster

Fairly early in the thesis work, I realized that my own computer would not have the computational capacities to do the number of simulations and tests I wanted to do. Consequently, I started to look at the possibility of using a high-performance computing cluster, access to which I was thankfully granted by the TU Delft.
Nonetheless, I faced major hurdles in taking advantage of this possibility. Here, the first hurdle was to learn to run programs on the cluster, which I could thankfully overcome with the help of some tutorials provided by the university. This also included my ability to install a newer version of python locally on the cluster without root access, as the version of python installed was too old to allow the use of tensorflow and keras, which I needed for implementing neural networks.
Next, I had to rewrite my code in such a way that it itself could take advantage of the multitude of processor cores available. Here, I had to work mainly on my own, as the method provided by the tutorials did no longer work as intended. Here, I must admit that I lost a week of my time unsuccessfully trying to use various tools which promised a minimal required change in coding. Unfortunately, these methods did not work, and I had to fall back onto using the mpi4py python library, which I had initially discarded as it required vast changes to the code I had already written.
A disadvantage of using mpi4py was that the easy training of neural network using the compile() and fit() functions from the keras library was no longer possible. Instead, each batch of training data had to be manually split up among the available core, which then calculated the gradients for each sample in regard to the trainable parameters of the network. All these gradients were then collected on a single core, where the network was updated. All updated parameters than had to be manually shared between cores so they could be updated on the other cores as well. Figuring out the best implementation with the least amount of overhead again took weeks.
Another problem regarding the cluster was my inability to access the powerful GPU available on the cluster. This was mainly due to the fact that there was no tutorial provided to enable this, and after failing to write a program which even recognized the GPU on the cluster, I gave up. In hindsight, this might have been a mistake, as accessing the GPU would likely have allowed me to significantly speed up my simulations (for examples in cases like optimizing c_1 from section 1.3, over 90% of the time is spent on training the neural network, due to the low evaluation time of the cost function).
The final problem regarding parallel programming were the problems I faced when debugging my code written for parallelization. The main problem here is that many errors which occurred during the execution of a program did not lead to its termination. This led to the situation that such errors (which might survive debugging on a smaller scale on my own computer) were not recognized until the computation time of the program grew far larger the previous estimations predicted. An example would be a capitalization mistake in a command designed to let different cores catch up with each other which led to the program waiting forever. As this did not result in an error message, I was only able to find the error after a week of debugging.

2.4 Personal Improvement

When looking back at the time spent on this master thesis, there are two different kinds of improvements I could recognize. The first one is obviously the increase in knowledge of different method and ways to implement them. At the forefront to mention here is certainly the topic of neural networks overall and

autoencoders specifically. This includes both recognizing how they are built, and how they work. For example, recognizing that neural networks (as long as one ignores more specific topics like recurrent neural networks) are basically a chain of parameterized functions, the training of which is basically just the solving of regression problem, took away much of the difficulties I had in comprehending them. Dealing with neural networks also gave me the ability to implement them using tools like tensorflow and keras.
Another point where I learned a lot is the field of parallel programming, especially when dealing with a distributed memory. For example, I learned how to achieve parallelization, while simultaneously minimizing the overhead required. Furthermore, I would say that my skill of using python in general improved as well, especially programming effectively, for example by eliminating as many for loops as possible, something I did not do as efficiently before starting my thesis. A further skill that could be mentioned here is me learning to produce high quality figures in latex using the tikzpicture and pfgplots libraries.
The second main improvement would in my opinion relate to my ability to manage a large research project. The first point to mention here would be my ability to do a literature review, where I in my opinion greatly improved when compared for example to the work I did during my bachelor thesis. Secondly, my ability to plan ahead, and building towards a final result also improved compared to my bachelor project.
But when looking back at the whole project, there are a lot of improvements the I should have made with hindsight. The main one is surely that I, despite recognizing pretty early that simple compliance minimization might be too well-behaved for it to be a good test for the proposed method, pressed on nonetheless. Instead, taking my time to find an equally cheap to evaluate, but far more non-convex topology optimization problem instead, would have prevent me from having to repeat simulations at the end of my thesis work. This mistake is especially egregious, as testing a cost function on non-convexity would have only required a simple test of random-local optimization. Another point, where I took not the best approach possible, is for example the failed use of the GPU on the cluster. Here, instead of just giving up, asking other people for their experience might have been advantageous, so I made sure I had exhausted every possible approach before abandoning this approach.

Further acknowledgments

While most of the work shown in this report was done by me on my own, a lot of different people helped me on my way to this final product, to whom I am greatly indebted.
First to mention here is surely Dr. Alejandro Aragón, who helped me along all the way, and who with his supervision and continuous helpful suggesting allowed me to make this realize the full potential of this work. His contribution is even more precious, if one considers that this project was completed during the unfortunately still ongoing pandemic, which deprived me of any personal contact.
I would also like to thank Dr. Matthijs Langelaar, for giving me access to the TU Delft high performance computing cluster and helping me to learn its usage by providing tutorials and being always ready to answer any question I got.
Furthermore, I would also like to thank my parents and my whole family in general for raising me in an environment which instilled into me a passion for engineering and mathematics which still persists to this day.
Lastly, I would also like to thank my friends here in Delft, and the local student orchestra, Krashna Musika, who made my stay in Delft worthwhile even outside the university.

References

[1] Isaac Newton. *Philosophiae naturalis principia mathematica*, volume 2. typis A. et JM Duncan, 1833.

[2] John Thøgersen. Unsustainable consumption. *European Psychologist*, 2014.

[3] Jeong Gil Cho, Jeong Seo Koo, and Hyun Seung Jung. A lightweight design approach for an emu carbody using a material selection method and size optimization. *Journal of Mechanical Science and Technology*, 30(2):673–681, 2016.

[4] GY Zhao, ZY Liu, Y He, HJ Cao, and YB Guo. Energy consumption in machining: Classification, prediction, and reduction strategy. *Energy*, 133:142–157, 2017.

[5] Diego M Jiménez-Bravo, Javier Pérez-Marcos, Daniel H De la Iglesia, Gabriel Villarrubia González, and Juan F De Paz. Multi-agent recommendation system for electrical energy optimization and cost saving in smart homes. *Energies*, 12(7), 2019.

[6] Feng Xiong, Xihong Zou, Zhigang Zhang, and Xiaohui Shi. A systematic approach for multi-objective lightweight and stiffness optimization of a car body. *Structural and Multidisciplinary Optimization*, 62(6):3229–3248, 2020.

[7] John Shalf. The future of computing beyond moore's law. *Philosophical Transactions of the Royal Society A*, 378(2166), 2020.

[8] Wen Long, Tiebin Wu, Ximing Liang, and Songjin Xu. Solving high-dimensional global optimization problems using an improved sine cosine algorithm. *Expert Systems with Applications*, 123:108–126, 2019.

[9] Jarosław Pillardy, Cezary Czaplewski, Adam Liwo, Jooyoung Lee, Daniel R. Ripoll, Rajmund Kaźmierkiewicz, Stanisław Ołdziej, William J. Wedemeyer, Kenneth D. Gibson, Yelena A. Arnautova, Jeff Saunders, Yuan-Jie Ye, and Harold A. Scheraga. Recent improvements in prediction of protein structure by global optimization of a potential energy function. *Proceedings of the National Academy of Sciences*, 98(5):2329–2333, 2001.

[10] Jui-Fang Chang and Peng Shi. Using investment satisfaction capability index based particle swarm optimization to construct a stock portfolio. *Information Sciences*, 181(14):2989–2999, 2011.

[11] Stephen Chen, James Montgomery, and Antonio Bolufé-Röhler. Measuring the curse of dimensionality and its effects on particle swarm optimization and differential evolution. *Applied Intelligence*, 42(3):514–526, 2015.

[12] D. Guirguis, N. Aulig, R. Picelli, B. Zhu, Y. Zhou, W. Vicente, F. Iorio, M. Olhofer, W. Matusik, C. A. Coello Coello, and K. Saitou. Evolutionary black-box topology optimization: Challenges and promises. *IEEE Transactions on Evolutionary Computation*, 24(4):613–633, 2020.

[13] Gilles Marck, Maroun Nemer, Jean-Luc Harion, Serge Russeil, and Daniel Bougeard. Topology optimization using the simp method for multiobjective conductive problems. *Numerical Heat Transfer, Part B: Fundamentals*, 61(6):439–470, 2012.

[14] Wenjie Zuo and Kazuhiro Saitou. Multi-material topology optimization using ordered simp interpolation. *Structural and Multidisciplinary Optimization*, 55(2):477–491, 2017.

[15] Hongliang Liu, Dixiong Yang, Peng Hao, and Xuefeng Zhu. Isogeometric analysis based topology optimization design with global stress constraint. *Computer Methods in Applied Mechanics and Engineering*, 342:625–652, 2018.

[16] Martin Philip Bendsøe and Ole Sigmund. *Topology optimization: theory, methods, and applications.* Springer Science & Business Media, 2003.

[17] G. I. N. Rozvany. A critical review of established methods of structural topology optimization. *Structural and Multidisciplinary Optimization*, 37:217–237, 01 2009.

[18] N van Dijk, M Langelaar, and F Van Keulen. Critical study of design parameterization in topology optimization; the influence of design parameterization on local minima. In *Proceedings of the 2nd International Conference on Engineering Optimization*, 2010.

[19] Anatoly Zhigljavsky and Antanas Žilinskas. *Bayesian and High-Dimensional Global Optimization.* Springer Nature, 2021.

[20] Mohamed Amine Bouhlel, Nathalie Bartoli, Rommel G Regis, Abdelkader Otsmane, and Joseph Morlier. Efficient global optimization for high-dimensional constrained problems by using the kriging models combined with the partial least squares method. *Engineering Optimization*, 50(12):2038–2053, 2018.

[21] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. Scalable global optimization via local bayesian optimization. *Advances in Neural Information Processing Systems*, 32:5496–5507, 2019.

[22] Iria CS Cosme, Isaac F Fernandes, João L de Carvalho, and Samuel Xavier-de Souza. Memory-usage advantageous block recursive matrix inverse. *Applied Mathematics and Computation*, 328:125–136, 2018.

[23] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[24] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[25] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[26] Stavros P Adam, Stamatios-Aggelos N Alexandropoulos, Panos M Pardalos, and Michael N Vrahatis. No free lunch theorem: A review. *Approximation and optimization*, pages 57–82, 2019.

[27] Wei Chu, Xiaogang Gao, and Soroosh Sorooshian. A new evolutionary search strategy for global optimization of high-dimensional problems. *Information Sciences*, 181(22):4909–4927, 2011.

[28] Yongjun Sun, Tong Yang, and Zujun Liu. A whale optimization algorithm based on quadratic interpolation for high-dimensional global optimization problems. *Applied Soft Computing*, 85, 2019.

[29] K. Kashima. Nonlinear model reduction by deep autoencoder of noise response data. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 5750–5755, 2016.

[30] D. Hartman and L. K. Mestha. A deep learning framework for model reduction of dynamical systems. In *2017 IEEE Conference on Control Technology and Applications (CCTA)*, pages 1917–1922, 2017.

[31] Danny D'Agostino, Andrea Serani, Emilio F Campana, and Matteo Diez. Deep autoencoder for off-line design-space dimensionality reduction in shape optimization. In *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2018.

[32] Kookjin Lee and Kevin T. Carlberg. Model reduction of dynamical systems on nonlinear manifolds using deep convolutional autoencoders. *Journal of Computational Physics*, 404, 2020.

[33] Gitta Kutyniok, Philipp Petersen, Mones Raslan, and Reinhold Schneider. A theoretical analysis of deep neural networks and parametric pdes. *Constructive Approximation*, pages 1–53, 2021.

[34] Romit Maulik, Bethany Lusch, and Prasanna Balaprakash. Reduced-order modeling of advection-dominated systems with recurrent neural networks and convolutional autoencoders. *Physics of Fluids*, 33(3):037106, 2021.

[35] Christodoulos A Floudas and Chrysanthos E Gounaris. A review of recent advances in global optimization. *Journal of Global Optimization*, 45(1):3–38, 2009.

[36] Padmavathi Kora and Priyanka Yadlapalli. Crossover operators in genetic algorithms: A review. *International Journal of Computer Applications*, 162(10), 2017.

[37] Rawaa Dawoud Al-Dabbagh, Ferrante Neri, Norisma Idris, and Mohd Sapiyan Baba. Algorithmic design issues in adaptive differential evolution schemes: Review and taxonomy. *Swarm and Evolutionary Computation*, 43:284–311, 2018.

[38] Dongshu Wang, Dapei Tan, and Lei Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22(2):387–408, 2018.

[39] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[40] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

[41] Luís Filipe Couto Azevedo Costa. Application of evolutionary swarms and autoencoders to wind-hydro coordination. 2008.

[42] V. Miranda, J. da Hora Martins, and V. Palma. Optimizing large scale problems with metaheuristics in a reduced space mapped by autoencoders—application to the wind-hydro coordination. *IEEE Transactions on Power Systems*, 29(6):3078–3085, 2014.

[43] Zhaoqi Gao, Zhibin Pan, Jinghuai Gao, and Zongben Xu. Building long-wavelength velocity for salt structure using stochastic full waveform inversion with deep autoencoder based model reduction. In *SEG Technical Program Expanded Abstracts 2019*, pages 1680–1684. Society of Exploration Geophysicists, 2019.

[44] Zhaoqi Gao, Chuang Li, Naihao Liu, Zhibin Pan, Jinghuai Gao, and Zongben Xu. Large-dimensional seismic inversion using global optimization with autoencoder-based model dimensionality reduction. *IEEE Transactions on Geoscience and Remote Sensing*, pages 1–15, 2020.

[45] Stephan Eismann, Stefan Bartzsch, and Stefano Ermon. Shape optimization in laminar flow with a label-guided variational autoencoder, 2017.

[46] Liwei Wang, Yu-Chin Chan, Faez Ahmed, Zhao Liu, Ping Zhu, and Wei Chen. Deep generative modeling for mechanistic-based learning and design of metamaterial systems. *Computer Methods in Applied Mechanics and Engineering*, 372, 2020.

[47] Jiaqi Jiang, David Sell, Stephan Hoyer, Jason Hickey, Jianji Yang, and Jonathan A Fan. Free-form diffractive metagrating design based on generative adversarial networks. *ACS nano*, 13(8):8872–8878, 2019.

[48] Zhaxylyk A. Kudyshev, Alexander V. Kildishev, Vladimir M. Shalaev, and Alexandra Boltasseva. Machine-learning-assisted metasurface design for high-efficiency thermal emitter optimization. *Applied Physics Reviews*, 7(2), 2020.

[49] Mauro Tucci, Sami Barmada, Luca Sani, Dimitri Thomopulos, and Nunzia Fontana. Deep neural networks based surrogate model for topology optimization of electromagnetic devices. In *2019 International Applied Computational Electromagnetics Society Symposium (ACES)*, pages 1–2, 2019.

[50] S Barmada, N Fontana, D Thomopulos, and M Tucci. Autoencoder based optimization for electromagnetics problems. *Applied Computational Electromagnetics Society Journal*, 34(12), 2019.

[51] Zijiang Yang, Xiaolin Li, L Catherine Brinson, Alok N Choudhary, Wei Chen, and Ankit Agrawal. Microstructural materials design via deep adversarial learning methodology. *Journal of Mechanical Design*, 140(11), 2018.

[52] Cem C. Tutum, Supawit Chockchowwat, Etienne Vouga, and Risto Miikkulainen. Functional generative design: an evolutionary approach to 3d-printing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1379–1386, 2018.

[53] Sangeun Oh, Yongsu Jung, Seongsin Kim, Ikjin Lee, and Namwoo Kang. Deep generative design: Integration of topology optimization and generative models. *Journal of Mechanical Design*, 141(11), 2019.

[54] Ryan-Rhys Griffiths and José Miguel Hernández-Lobato. Constrained bayesian optimization for automatic chemical design using variational autoencoders. *Chemical science*, 11(2):577–586, 2020.

[55] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[56] Ole Sigmund. A 99 line topology optimization code written in matlab. *Structural and multidisciplinary optimization*, 21(2):120–127, 2001.

[57] John A. Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[58] Ioannis PA Papadopoulos, Patrick E Farrell, and Thomas M Surowiec. Computing multiple solutions of topology optimization problems. *SIAM Journal on Scientific Computing*, 43(3):A1555–A1582, 2021.

[59] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

[60] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, pages 2672–2680. 2014.

[61] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. Adversarial autoencoders, 2016.

[62] Laith Abualigah, Ali Diabat, Seyedali Mirjalili, Mohamed Abd Elaziz, and Amir H Gandomi. The arithmetic optimization algorithm. *Computer methods in applied mechanics and engineering*, 376, 2021.

[63] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[64] Martin Philip Bendsøe and Noboru Kikuchi. Generating optimal topologies in structural design using a homogenization method. *Computer Methods in Applied Mechanics and Engineering*, 71(2):197–224, 11 1988.

[65] Martin Philip Bendsøe. Optimal shape design as a material distribution problem. *Structural Optimization*, 1(4):193–202, 1989.

[66] Erik Andreassen, Anders Clausen, Mattias Schevenels, Boyan S Lazarov, and Ole Sigmund. Efficient topology optimization in matlab using 88 lines of code. *Structural and Multidisciplinary Optimization*, 43(1):1–16, 2011.

[67] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE transactions on evolutionary computation*, 15(1):4–31, 2010.

[68] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[69] David JJ Toal, Neil W Bressloff, and Andy J Keane. Kriging hyperparameter tuning strategies. *AIAA journal*, 46(5):1240–1252, 2008.

[70] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. Scalable bayesian optimization using deep neural networks. In *International conference on machine learning*, pages 2171–2180, 2015.

[71] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for bayesian optimization. In *UAI*, pages 327–336. Citeseer, 2011.

[72] Egbert JW Boers and Herman Kuiper. Biological metaphors and the design of modular artificial neural networks. 1992.

[73] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[74] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel Distributed Processing*, 1, 1986.

[75] Mark A Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991.

[76] David Charte, Francisco Charte, Salvador García, María J del Jesus, and Francisco Herrera. A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines. *Information Fusion*, 44:78–96, 2018.

[77] Alireza Makhzani and Brendan J Frey. Winner-take-all autoencoders. In *Advances in Neural Information Processing Systems 28*, pages 2791–2799. 2015.

[78] Arnab Mondal, Sankalan Pal Chowdhury, Aravind Jayendran, Himanshu Asnani, Parag Singla, and AP Prathosh. Maskaae: Latent space optimization for adversarial auto-encoders. In *Conference on Uncertainty in Artificial Intelligence*, pages 689–698. PMLR, 2020.

[79] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 165–174, 2019.

[80] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[81] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[82] Yingbo Zhou, Devansh Arpit, Ifeoma Nwogu, and Venu Govindaraju. Is joint training better for deep auto-encoders?, 2015.

[83] M. R. Spiegel, J. J. Schiller, R. A. Srinivasan, and Mike LeVan. *Probability and statistics*, volume 2. 2001.

[84] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. 2015.

[85] Larry Wasserman. *All of nonparametric statistics.* Springer Science & Business Media, 2006.

[86] Alejandro M Aragón and Angelo Simone. The discontinuity-enriched finite element method. *International Journal for Numerical Methods in Engineering*, 112(11):1589–1613, 2017.

A Computational Methods

A.1 Adam

Adam [55] is a local optimization method using gradient descent with momentum. Hereby, the updated variables $\boldsymbol{\theta}_{i+1}$ are generated based on the previous ones $\boldsymbol{\theta}_i$ and the gradient of the cost function $\boldsymbol{g}_i = \nabla_{\boldsymbol{\theta}} c(\boldsymbol{\theta}_i)$. Adam additionally uses a number of vectors to memorize previous results, avoiding a very erratic search pattern [55], with $\odot$ being the Hadamard product:

$$\begin{aligned} \boldsymbol{m}_i &= \beta_1 \boldsymbol{m}_{i-1} + (1-\beta_1)\boldsymbol{g}_i \\ \boldsymbol{v}_i &= \beta_2 \boldsymbol{v}_{i-1} + (1-\beta_2)\boldsymbol{g}_i \odot \boldsymbol{g}_i \\ \widehat{\boldsymbol{m}}_i &= \frac{\boldsymbol{m}_i}{1-\beta_1^i} \\ \widehat{\boldsymbol{v}}_i &= \frac{\boldsymbol{v}_i}{1-\beta_2^i} \\ [\theta_{i+1}]_j &= [\theta_i]_j - \frac{\alpha}{\left(\sqrt{[\widehat{v}_i]_j} + \epsilon\right)} [\widehat{m}_i]_j \end{aligned} \tag{10}$$

The learning rate α and the decay rates β_1 and β_2 are used to tune the process. The $\epsilon = 10^{-6}$ is there to prevent divisions by zero.

A.2 SIMP

The solid isoparametric material with penalization method (SIMP) is a method used for gradient based topology optimization, first proposed by Bendsøe in the late 1980s [64, 65]. In this work, it is used to solve compliance minimization problems:

$$\begin{aligned} \min_{\boldsymbol{x} \in [0,1]^{n_y \times n_x}} & C(\boldsymbol{x}) = \boldsymbol{u}(\boldsymbol{x})^T \boldsymbol{F} \\ \boldsymbol{K}(\boldsymbol{x})\boldsymbol{u}(\boldsymbol{x}) &= \boldsymbol{F} \\ \overline{\boldsymbol{x}} &\overset{!}{=} \rho_0 = 0.4 \end{aligned} \tag{11}$$

Here, $\boldsymbol{K}$ is the stiffness matrix resulting from the mesh, and $\boldsymbol{F}$ is the force vector, while $\boldsymbol{u}$ is the resulting displacement field.
SIMP can now be used to advance from an initial material distribution $\boldsymbol{x_0}$ to a final distribution $\boldsymbol{x}_\lambda$. Here, each element of the density field $\boldsymbol{x_i}$ has to be larger than zero ($[x_i]_{k,l} \in [0.0001, 1]$). To get from $\boldsymbol{x}_i$ to $\boldsymbol{x}_{i+1}$, the following iterative step has to be taken:

- The stiffness matrix $\boldsymbol{K}_i = \boldsymbol{K}(\boldsymbol{x}_i)$ has to be constructed. Here, for each element one assumes a Poisson's ration $\nu_{k,l} = \nu_0 = 0.3$ and a Young's modulus of $[E_i]_{k,l}$, with the penalization of $p = 3$ and $E_0 = 1$.

  $$[E_i]_{k,l} = E_0 \left(\frac{1}{10000} + \frac{9999}{10000} \left([x_i]_{k,l} \right)^p \right) \tag{12}$$

  One can then find the displacement $\boldsymbol{u}_i$ with $\boldsymbol{K}_i \boldsymbol{u}_i = \boldsymbol{F}$. This allows the calculation of the compliance $C_i = \boldsymbol{u}_i^T \boldsymbol{F}$

- The sensitivity $\boldsymbol{S}_{C,i}$ has to be calculated:

  $$\begin{aligned} [S_{C,i}]_{k,l} &= -\frac{\partial C_i}{\partial [x_i]_{k,l}} \\ &= -\frac{9999}{10000} E_0 p \left([x_i]_{k,l} \right)^{p-1} \boldsymbol{u}_i^T \frac{\partial \boldsymbol{K}_i}{\partial [E_i]_{k,l}} \boldsymbol{u}_i \end{aligned} \tag{13}$$

  For details on the assembly of $\boldsymbol{K}_i$ and the derivative $\partial \boldsymbol{K}_i / \partial [E_i]_{k,l}$ see Andreassen et al. [66].

- If only the sensitivity is used to update the design, then SIMP often leads to a checkerboard pattern. As such a design is not manufacturable, one of the following methods can be used to update the sensitivity to $\boldsymbol{G}_i$:
  - On can use a filter on the designs, which is characterized by the filter radius $r_{\min} > 1$. The implementation can be seen in Andreassen et al. [66].
  - By setting $r_{\min} \in (0, 1]$, the filter will have no effect. In this case, one can add the sensitivity of the Ginzburg-Landau energy term to $\boldsymbol{S}_{C,i}$ to avoid the checkerboarding (see Papadopoulos et al. [58]):

$$\begin{aligned}[G_i]_{k,l} = & [S_{C,i}]_{k,l}[x_i]_{k,l} + \frac{\beta}{\epsilon}\left(1 - 2[x_i]_{k,l}\right) \\ & + \beta\epsilon\left(4[x_i]_{k,l} - [x_i]_{k+1,l} - [x_i]_{k-1,l} - [x_i]_{k,l+1} - [x_i]_{k,l-1}\right)\end{aligned} \tag{14}$$

  Here, the parameters $\beta = 0.05$ and ϵ are used to tune the process.

- Lastly, one has to advance. For this, one calculates $\boldsymbol{D}_i$:

$$[D_i]_{k,l} = \sqrt{[G_i]_{k,l}[x_i]_{k,l}} \tag{15}$$

  One than start an iterative process, with $l_{l,0} = 0$ and $l_{u,1} = 1000000$:

$$\begin{aligned}[x_{i+1}]_{k,l} &= \max\left\{\min\left\{\frac{2[D_i]_{k,l}}{l_{l,j} + l_{u,j}}, [x_i]_{k,l} + 0.25, 1\right\}, [x_i]_{k,l} - 0.25, 0.0001\right\} \\ \{l_{l,j+1}, l_{u,j+1}\} &= \begin{cases}\left\{l_{l,j}, \frac{l_{l,j}+l_{u,j}}{2}\right\} & \overline{\boldsymbol{x}}_{i+1} \leq \rho_0 \\ \left\{\frac{l_{l,j}+l_{u,j}}{2}, l_{u,j}\right\} & \overline{\boldsymbol{x}}_{i+1} > \rho_0\end{cases}\end{aligned} \tag{16}$$

  This will be done while the following condition is fulfilled, after which one has the next step $\boldsymbol{x}_{i+1}$:

$$\frac{l_{u,j} - l_{l,j}}{l_{u,j} + l_{l,j}} > 0.0001 \tag{17}$$

Due to the penalization $p > 1$, using intermediate values for the density is disadvantageous, and the final outcome will be likely sparse. This is a possible method to allow for manufacturable designs to be produced.

A.3 Differential Evolution

Differential evolution [23] is a population based method which works over multiple generations, with each population $\boldsymbol{P}_g$ at generation $g \in \{0, \ldots, G\}$ consisting out of γ individuals:

$$\boldsymbol{P}_g = \{\boldsymbol{p}_{1,g}, \ldots, \boldsymbol{p}_{\gamma,g}\}, \ \boldsymbol{p}_{1,g} \in Z \tag{18}$$

$\boldsymbol{P}_0$ will be randomly generated, but afterwards, the following generation will be created with three steps, which are performed on every individual $\boldsymbol{p}_{i,g}$.

1. Firstly, a child $\widehat{\boldsymbol{p}}_{i,g}$ will be created for the parent $\boldsymbol{p}_{i,g}$:

$$\widehat{\boldsymbol{p}}_{i,g} = \boldsymbol{p}_{a,g} + F(\boldsymbol{p}_{b,g} - \boldsymbol{p}_{c,g}) \tag{19}$$

   Here, $a, b, c \in \{1, \ldots, \gamma\}$ are randomly chosen variables, while F is a hyperparameter that has to be chosen at the beginning of the process. While Storn and Price [23] suggest that i, a, b, and c should be all different from each other, this was not actively enforced in this work to achieve faster computation time, as even with the smallest γ used in this work ($\gamma = 100$), the probability of this condition being fulfilled seems high enough to not negatively influence the process overall:

$$P(a \neq b \neq c \neq i) = \frac{\gamma - 1}{\gamma}\frac{\gamma - 2}{\gamma}\frac{\gamma - 3}{\gamma} \gtrsim 0.941 \ \ \forall \gamma \geq 100 \tag{20}$$

It has to be noted that $\widehat{\boldsymbol{p}}_{i,g} \notin Z$ is possible. Therefore, in every dimensions j of $\widehat{\boldsymbol{p}}_{i,g}$, a clipping function is used to enforce the boundaries of z, under the assumption that Z is a hyperrectangle:

$$\left[\widehat{p}_{i,g}\right]_j = \begin{cases} z_{\min,j} & \left[\widehat{p}_{i,g}\right]_j < z_{\min,j} \\ \left[\widehat{p}_{i,g}\right]_j & z_{\min,j} \leq \left[\widehat{p}_{i,g}\right]_j < z_{\max,j} \\ z_{\max,j} & z_{\max,j} < \left[\widehat{p}_{i,g}\right]_j \end{cases} \tag{21}$$

2. After the child $\widehat{\boldsymbol{p}}_{i,g}$ of the parent $\boldsymbol{p}_{i,g}$ has been build, a binary crossover between them will be done, the result being $\widetilde{\boldsymbol{p}}_{i,g}$. Here, for each element, a random number $\chi_{j,i,g} \sim U(0,1)$ will be needed. One than gets the crossover in the following way, where χ_0 is another hyperparameter:

$$\left[\widetilde{p}_{i,g}\right]_j = \begin{cases} \left[\widehat{p}_{i,g}\right]_j & \chi_{j,i,g} < \chi_0 \\ \left[p_{i,g}\right]_j & \chi_{j,i,g} \geq \chi_0 \end{cases} \tag{22}$$

3. Lastly, one has to select the individual to survive to the next generation:

$$\boldsymbol{p}_{i,g+1} = \begin{cases} \boldsymbol{p}_{i,g} & c_\mu\left(\widetilde{\boldsymbol{p}}_{i,g}\right) > c_\mu\left(\boldsymbol{p}_{i,g}\right) \\ \widetilde{\boldsymbol{p}}_{i,g} & c_\mu\left(\widetilde{\boldsymbol{p}}_{i,g}\right) \leq c_\mu\left(\boldsymbol{p}_{i,g}\right) \end{cases} \tag{23}$$

Overall, four parameters can be used to tune the whole process, namely the population size γ, the number of generations G, the factor F and the probability χ_0. The whole process than can be defined as:

$$\boldsymbol{P}_G = DE_{\gamma,G,F,\chi_0}(\boldsymbol{P}_0) \tag{24}$$

In some cases, the process can include the selection of the best member of $\boldsymbol{P}_G$ as the final output. It has also to be noted that there are more advanced version of differential evolution [67], which are not used in this work.

A.4 Bayesian Optimization

Bayes optimization [68] is a method for global optimization, used mainly in the case of very expensive cost functions c, as it requires less evaluations of this cost function c. It is a process with multiple iterations $i \in \{0, \dots, i_{BO}\}$, where each iteration consists out of three steps.

- In a first step, a response surface model will be built, based upon M_i sampled points $\boldsymbol{z}_{i,j} \in Z_i$ in latent space ($\boldsymbol{Z_i} = \{\boldsymbol{z}_{1,i}, \dots, \boldsymbol{z}_{M_i,i}\}$). For each of these points, the cost function c will be evaluated, forming the vector $\boldsymbol{c}_i$. Based upon $\boldsymbol{Z}_i$, one then has to determine the correlation matrix $\boldsymbol{R}_{\boldsymbol{\theta},\boldsymbol{\rho}} \in \mathbb{R}^{M_i \times M_i}$, with:

$$\left[\boldsymbol{R}_{\boldsymbol{\theta},\boldsymbol{\rho},i}\right]_{k,l} = \exp\left(-\sum_{o=1}^{m} \theta_o \left|\left[z_{k,i}\right]_o - \left[z_{l,i}\right]_o\right|^{\rho_o}\right), \quad \forall\, o:\ \theta_o \in [0.1, 5],\ \rho_o \in [0.6, 5.5] \tag{25}$$

  Here, $\boldsymbol{\theta}, \boldsymbol{\rho} \in \mathbb{R}_+^m$ are parameters that can be tuned to fit the model to the data. Now one can find $\boldsymbol{\theta}_i^*, \boldsymbol{\rho}_i^*$, with:

$$\begin{aligned} \boldsymbol{\theta}_i^*, \boldsymbol{\rho}_i^* &= \underset{\boldsymbol{\theta},\boldsymbol{\rho}}{\operatorname{argmax}}\ N\left(\boldsymbol{c}_i \,|\, \boldsymbol{1}\mu_{\boldsymbol{\theta},\boldsymbol{\rho},i}, \boldsymbol{R}_{\boldsymbol{\theta},\boldsymbol{\rho},i}\sigma^2_{\boldsymbol{\theta},\boldsymbol{\rho},i}\right) \\ \mu_{\boldsymbol{\theta},\boldsymbol{\rho},i} &= \frac{\boldsymbol{1}^T \boldsymbol{R}^{-1}_{\boldsymbol{\theta},\boldsymbol{\rho},i}\boldsymbol{c}_i}{\boldsymbol{1}^T \boldsymbol{R}^{-1}_{\boldsymbol{\theta},\boldsymbol{\rho},i}\boldsymbol{1}} \\ \sigma_{\boldsymbol{\theta},\boldsymbol{\rho},i} &= \sqrt{\frac{\left(\boldsymbol{c}_i - \boldsymbol{1}\mu_{\boldsymbol{\theta},\boldsymbol{\rho},i}\right)^T \boldsymbol{R}^{-1}_{\boldsymbol{\theta},\boldsymbol{\rho},i}\left(\boldsymbol{c}_i - \boldsymbol{1}\mu_{\boldsymbol{\theta},\boldsymbol{\rho},i}\right)}{M_i}} \end{aligned} \tag{26}$$

  where N is the multivariate normal distribution and $\boldsymbol{1} \in \mathbb{R}^{M_i \times 1}$ is a vector with every element being 1. This can be simplified using a logarithmic transformation [69]:

$$\boldsymbol{\theta}_i^*, \boldsymbol{\rho}_i^* = \underset{\boldsymbol{\theta},\boldsymbol{\rho}}{\operatorname{argmin}}\ \left(M_i \ln\left(\sigma^2_{\boldsymbol{\theta},\boldsymbol{\rho},i}\right) + \ln\left(\left|\boldsymbol{R}_{\boldsymbol{\theta},\boldsymbol{\rho},i}\right|\right)\right) \tag{27}$$

Here, $\mu_i = \mu_{\boldsymbol{\theta}_i^*,\boldsymbol{\rho}_i^*,i}$, $\sigma_i = \sigma_{\boldsymbol{\theta}_i^*,\boldsymbol{\rho}_i^*,i}$, and $\boldsymbol{R}_i = \boldsymbol{R}_{\boldsymbol{\theta}_i^*,\boldsymbol{\rho}_i^*,i}$ is then defined.
As this optimization problem (equation (27)) has to be solved for every iteration step, there are two methods used. The first one employs differential evolution $DE_{100,500,0.6,0.9}$. In the second one, Adam with $\alpha = 0.005$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ is used for 10 iterations. According to recommendations in [69], the first model $\boldsymbol{R}_0$, as well as the last one $\boldsymbol{R}_{100}$ will be build using differential evolution $DE_{100,100,0.9,0.6}$. Additionally, every $\boldsymbol{R}_{hT}$ with $h \in \mathbb{N}$ will also use differential evolution. The remaining iterations will on the other hand use Adam to update the underlying model.
Based on $\boldsymbol{R}_i$, the approximation function $\widehat{c}_i$[68] can be calculated:

$$\begin{aligned}\widehat{c}_i(\boldsymbol{z}) &= \mu_i + \boldsymbol{r}_i(\boldsymbol{z})^T \boldsymbol{R}_i^{-1} (\boldsymbol{c}_i - \boldsymbol{1}\mu_i) \\ [\boldsymbol{r}_i(\boldsymbol{z})]_k &= \exp\left(-\sum_{o=1}^{m} [\theta_i^*]_o \left|z_o - [z_{k,i}]_o\right|^{[\rho_i^*]_o}\right), \quad \boldsymbol{r}_i(\boldsymbol{z}) \in \mathbb{R}^{M_i \times 1}\end{aligned} \tag{28}$$

This approach also allows to predict the uncertainty of the surrogate model, here depicted by the mean squared error $s(\boldsymbol{z})^2$:

$$s_i(\boldsymbol{z})^2 = \sigma_i^2 \left(1 - \boldsymbol{r}_i(\boldsymbol{z})^T \boldsymbol{R}_i^{-1} \boldsymbol{r}_i(\boldsymbol{z}) + \frac{1 - \boldsymbol{1}^T \boldsymbol{R}_i^{-1} \boldsymbol{r}_i(\boldsymbol{z})}{\boldsymbol{1}^T \boldsymbol{R}_i^{-1} \boldsymbol{1}}\right) \tag{29}$$

This step, the creation of the surrogate model, is a cubic problem [70]. Therefore, while a higher M_i might provide a more accurate model, increasing M_i would lead to a significant increase in computational expense.

- The second step would then be the optimization of a so called acquisition function. In building such a function one must find a compromise between exploration (look at regions with few available information) and exploitation (look at regions with already known good cost function values). The acquisition function used here is the expected improvement function EI, proposed by Jones et al. [68].

$$EI_i(\boldsymbol{z}) = (c_{i,\min} - \widehat{c}_i(\boldsymbol{z}) - \xi_i)\, \Phi\left(\frac{c_{i,\min} - \widehat{c}_i(\boldsymbol{z}) - \xi_i}{s_i(\boldsymbol{z})}\right) + s_i(\boldsymbol{z}) \phi\left(\frac{c_{i,\min} - \widehat{c}_i(\boldsymbol{z}) - \xi_i}{s_i(\boldsymbol{z})}\right) \tag{30}$$

Here, $\Phi(x)$ is the cumulative distribution function of the normal distribution $N(x|0,1) = \phi(x)$, and $c_{i,\min}$ is the minimal element in the vector $\boldsymbol{c}_i$. Meanwhile, ξ_i is a factor which emphasizes exploration in the beginning of the optimization process and exploitation at the end [71]. When calculating the acquisition function, a small margin (10^{-8}) will be added to the denominators to avoid division by zero.
EI_i then must be maximized, resulting in the optimized point $\boldsymbol{z}_i^*$.

- The sample points than have to be updated:

$$\begin{aligned}\boldsymbol{Z}_{i+1} &= \boldsymbol{Z}_i \cup \{\boldsymbol{z}_i^*\} \\ \boldsymbol{c}_{i+1} &= \boldsymbol{c}_i \cup \{c(\boldsymbol{z}_i^*)\}\end{aligned} \tag{31}$$

Then, the optimum will be determined as the member of $\boldsymbol{Z}_{i_{BO}}$ with the lowest cost function.
Overall, this process reduces the number of evaluations of the cost function c to $M_0 + i_{BO}$, which might reduce the computation time compared to other methods, if $\boldsymbol{c}$ is sufficiently expensive.
Overall, four parameters can be used to tune the whole process, namely the initial number of random samples M_0, the number of iterations i_{BO}, the frequency of full model updates T, and ξ_i. Additionally, the method used to maximize EI_i can have an influence on the overall performance of Bayesian optimization.

A.5 Feed-forward Neural Networks

An Artificial Neural Network is in its most basic form simply a graph of connected nodes, which in some capacity tries to model the behavior of human brains [72]. In the case of a feed-forward network, these nodes, which are also known as neurons, are each part of different layers $\boldsymbol{x}^{(l)}$ with $l \in \{0, \ldots, L\}$. $\boldsymbol{x}^{(0)}$ is commonly called the input layer, while $\boldsymbol{x}^{(0)}$ is referred to as output layer. Layers in between are known as hidden layers. These layers are connected by often nonlinear functions $f^{(l)}_{\boldsymbol{\theta}^{(l)}}$, which are dependent on a number of parameters $\boldsymbol{\theta}^{(l)}$.

$$\boldsymbol{x}^{(l)} = f^{(l)}_{\boldsymbol{\theta}^{(l)}}\left(\boldsymbol{x}^{(l-1)}\right) \tag{32}$$

Consequently, a feed-forward neural network can also be seen as a number of function compositions:

$$\boldsymbol{x}^{(L)} = \left(f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \ldots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}\right)\left(\boldsymbol{x}^{(0)}\right) \tag{33}$$

There can be different kinds of functions used for this [73].

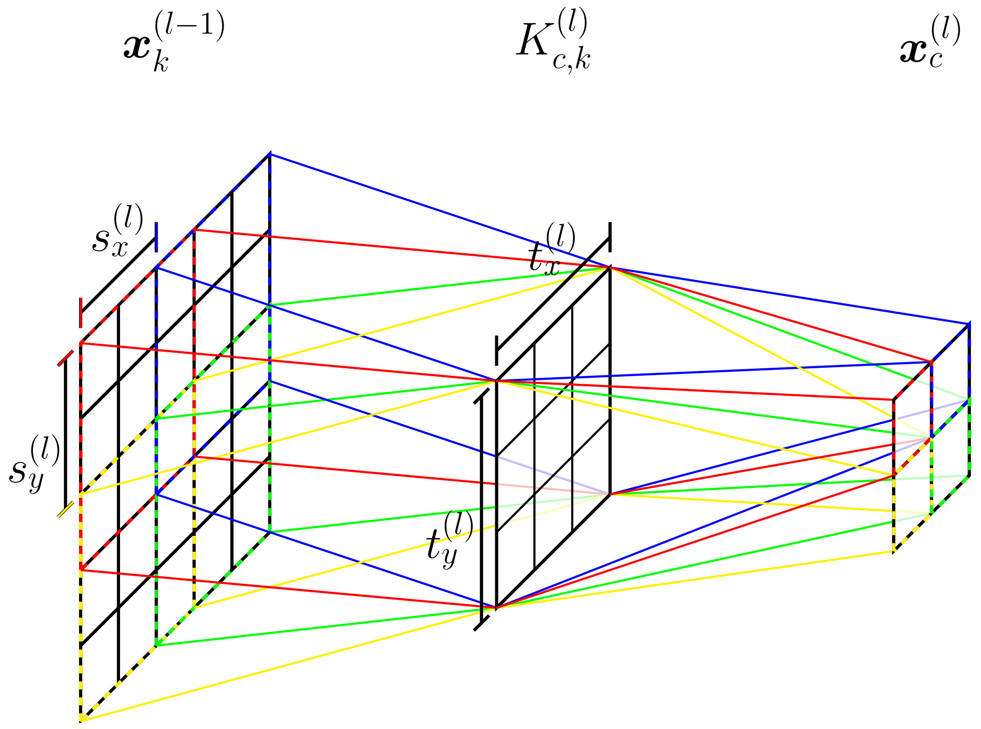


Figure 6: Representation of a convolutional connection between the channels $\boldsymbol{x}_k^{(l-1)}$ and $\boldsymbol{x}_c^{(l)}$, using the filter $K_{c,k}^{(l)}$, and a stride $s_y^{(l)} = s_x^{(l)} = 2$, as well as a filter size $t_y^{(l)} = t_x^{(l)} = 3$. In this work, $s_y^{(l)} = s_x^{(l)} = s^{(l)}$ and $t_y^{(l)} = t_x^{(l)} = t^{(l)}$ will always be the case.
Here, $\widehat{\boldsymbol{x}}_{1,1,k}^{(l-1)}$ and $x_{1,1,c}^{(l)}$ are seen in red boundaries in $\boldsymbol{x}_k^{(l-1)}$ and $\boldsymbol{x}_c^{(l)}$ respectively. Similarly, $\widehat{\boldsymbol{x}}_{2,1,k}^{(l-1)}$ and $x_{2,1,c}^{(l)}$ are surrounded in yellow, $\widehat{\boldsymbol{x}}_{1,2,k}^{(l-1)}$ and $x_{1,2,c}^{(l)}$ in blue, and $\widehat{\boldsymbol{x}}_{2,2,k}^{(l-1)}$ and $x_{2,2,c}^{(l)}$ in green.

- One possible type of function used are dense connections ($\mathbb{O}_\Delta$) between layers. In this case, each element (also called neuron) $x_i^{(l-1)}$ with $i \in \{1, \ldots, \nu^{(l)}\}$ of layer $\boldsymbol{x}^{(l-1)} \in \mathbb{R}^{\nu^{(l)}}$ is connected to every neuron $x_j^{(l)}$ of the layer $\boldsymbol{x}^{(l)}$ by the weight $w_{j,i}^{(l)}$ ($\boldsymbol{W}^{(l)} \in \mathbb{R}^{\nu^{(l)} \times \nu^{(l-1)}}$). Additionally, each neuron $x_j^{(l)}$ has a bias value $b_j^{(l)}$ ($\boldsymbol{b}^{(l)} \in {}^{\nu^{(l)}}$) as well as an activation function $\sigma^{(l)}$:

$$\boldsymbol{x}^{(l)} = f^{(l)}_{\boldsymbol{\theta}^{(l)}}\left(\boldsymbol{x}^{(l-1)}\right) = \sigma^{(l)}\left(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)}\right) \tag{34}$$

  One therefore has the parameters $\boldsymbol{\theta}^{(l)} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}$.

- A different type of function is the convolutional connection ($\mathbb{O}_C$). Instead of the dense layer, which is mainly used to connect two one-dimensional vectors, a convolutional function in this work connects two layers consisting of three-dimensional tensors. Each neuron $x_{i,j,k}^{(l)}$ is part of row $i \in \{1, \ldots, \nu_y^{(l)}\}$,

column $j \in \{1, \dots, \nu_x^{(l)}\}$, as well as channel $k \in \{1, \dots, \kappa^{(l)}\}$, with $\boldsymbol{x}^{(l)} \in \mathbb{R}^{\nu_y^{(l)} \times \nu_x^{(l)} \times \kappa^{(l)}}$.
In a convolutional connection, each channel k of the layer $\boldsymbol{x}^{(l-1)}$ is connected to every channel c of the layer $\boldsymbol{x}^{(l)}$ by a filter $\boldsymbol{K}_{c,k}^{(l)} \in \mathbb{R}^{t_y^{(l)} \times t_x^{(l)}}$. Here, $t_y^{(l)} \times t_x^{(l)}$ is the filter size. While rectangular filters are possible, it is common to use quadratic filters ($t_y^{(l)} = t_x^{(l)} = t^{(l)}$), which is done often in this work as well.
For every node $x_{i,j,c}^{(l)}$, a part of $\boldsymbol{x}_k^{(l-1)}$, namely $\widehat{\boldsymbol{x}}_{i,j,k}^{(l-1)}$ (chosen depending on filter size $t^{(l)}$ and stride $s^{(l)}$, see Figure 6), has this filter applied. The results of this for all k are added, and a bias $b_{i,j,c}^{(l)}$ is applied as well as an activation function ($\langle \cdot , \cdot \rangle_F$ is the Frobenius inner product):

$$x_{i,j,c}^{(l)} = \sigma^{(l)} \left(b_c^{(l)} + \sum_{k=1}^{\kappa^{(l-1)}} \langle \widehat{\boldsymbol{x}}_{i,j,k}^{(l-1)}, \boldsymbol{K}_{c,k}^{(l)} \rangle_F \right) \tag{35}$$

Consequently, the parameters $\boldsymbol{\theta}^{(l)} = \{\boldsymbol{K}^{(l)}, \boldsymbol{b}^{(l)}\}$ define the function $f_{\boldsymbol{\theta}^{(l)}}^{(l)}$, with $\boldsymbol{K}^{(l)} \in \mathbb{R}^{\kappa^{(l)} \times \kappa^{(l-1)} \times t_y^{(l)} \times t_x^{(l)}}$.

- As similar layer to the convolutional connection is the max pooling connection ($\mathbb{O}_M$). Depending on filter sizes $t_y^{(l)}$ and $t_x^{(l)}$ and stride $s^{(l)}$, $\widehat{\boldsymbol{x}}_{i,j,k}^{(l-1)}$ is chosen. One can then determine:

$$x_{i,j,c}^{(l)} = \max \left(\widehat{\boldsymbol{x}}_{i,j,k}^{(l-1)} \right) \tag{36}$$

  In this case, the function $f_{\boldsymbol{\theta}^{(l)}}^{(l)}$ has an empty parameter set $\boldsymbol{\theta}^{(l)} = \{\ \}$. The activation function used here is normally the linear one, so it has not been depicted in the above equation.

- Another connection used in this work is the deconvolutional connection ($\mathbb{O}_D$). Here, similarly to $\widehat{\boldsymbol{x}}_{i,j,k}^{(l-1)}$ and $x_{i,j,c}^{(l)}$ in the previous cases, $\widehat{\boldsymbol{x}}_{i,j,c}^{(l)}$ can be determined for a node $x_{i,j,k}^{(l-1)}$, based on filter sizes $t_y^{(l)}$ and $t_x^{(l)}$ and stride $s^{(l)}$:

$$\widehat{\boldsymbol{x}}_{i,j,c}^{(l)} = \sum_{k=1}^{\kappa^{(l-1)}} \boldsymbol{K}_{c,k}^{(l)} x_{i,j,k}^{(l-1)} \tag{37}$$

  Finally, the channel $\boldsymbol{x}_c^{(l)}$ has to be assembled, with $i \in \{1, \dots, \nu_y^{(l-1)}\}$ and $j \in \{1, \dots, \nu_x^{(l-1)}\}$:

$$\boldsymbol{x}_c^{(l)} = \sigma^{(l)} \left(\boldsymbol{b}_c^{(l)} + \mathop{\mathbb{A}}_{i,j} \widehat{\boldsymbol{x}}_{i,j,c}^{(l)} \right) \tag{38}$$

  It has to be noted that the biases inside $\boldsymbol{b}_c$ will always be the same value. The parameters for this function are $f_{\boldsymbol{\theta}^{(l)}}^{(l)}$ are $\boldsymbol{\theta}^{(l)} = \{\boldsymbol{K}^{(l)}, \boldsymbol{b}^{(l)}\}$.
  If one chooses, as was done in this work, $\nu_x^{(l-1)} = s_x^{(l)}(\nu_x^{(l)} - 1) + t_x^{(l)}$ and $\nu_y^{(l-1)} = s_y^{(l)}(\nu_y^{(l)} - 1) + t_y^{(l)}$ for convolutional or max pooling connections, as well as $\nu_x^{(l)} = s_x^{(l)}(\nu_x^{(l-1)} - 1) + t_x^{(l)}$ and $\nu_y^{(l)} = s_y^{(l)}(\nu_y^{(l-1)} - 1) + t_y^{(l)}$ for deconvolutional connections, padding issues can be completely avoided.

- Next, one also need a connection that allows a transformation to flatten ($\mathbb{O}_F$) a layer from a three-dimensional tensor to an one-dimensional representation ($\nu_y^{(l-1)}\nu_x^{(l-1)}\kappa^{(l-1)} = \nu^{(l)}$):

$$\mathbb{O}_F = f^{(l)} : \mathbb{R}^{\nu_y^{(l-1)} \times \nu_x^{(l-1)} \times \kappa^{(l-1)}} \to \mathbb{R}^{\nu_y^{(l-1)}\nu_x^{(l-1)}\kappa^{(l-1)}} \tag{39}$$

- Lastly, on also needs a connection which can transform a one dimensional layer back into a three dimensional one, which can be considered as an inversion of the flattening connection $\mathbb{O}_F$, with $\nu_y^{(l)}\nu_x^{(l)}\kappa^{(l)} = \nu^{(l-1)}$ being a condition to be fullfilled:

$$\mathbb{O}_I(\nu_y^{(l)}, \nu_x^{(l)}, \kappa^{(l)}) = f^{(l)} : R^{\nu^{(l-1)}} \to \mathbb{R}^{\nu_y^{(l)} \times \nu_x^{(l)} \times \kappa^{(l)}} \tag{40}$$

$$\left(\mathbb{O}_I(\nu_y^{(l)}, \nu_x^{(l)}, \kappa^{(l)}) \circ \mathbb{O}_F \right) \left(\boldsymbol{x}^{(l)} \right) = \boldsymbol{x}^{(l)} \tag{41}$$

In this work, four different ativation function are then used:

$$\begin{aligned}
\sigma_L(x) &= x \\
\sigma_S(x) &= \frac{1}{1+\exp(-x)} \\
\sigma_T(x) &= \tanh(x) \\
\sigma_R(x) &= \begin{cases} x & x \geq 0 \\ 0.3x & x < 0 \end{cases}
\end{aligned} \tag{42}$$

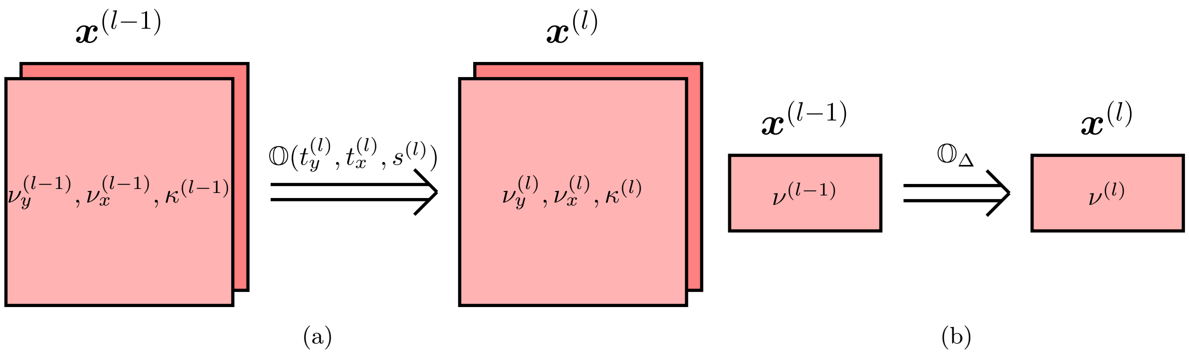


Figure 7: Symbolic representation of different connections between layers. In (a), the depiction is of either convolutional ($\mathbb{O} = \mathbb{O}_C$), max pooling ($\mathbb{O} = \mathbb{O}_M$), or deconvolutional ($\mathbb{O} = \mathbb{O}_D$) connections. In (b), the depiction is of a dense connection, marked with $\mathbb{O}_\Delta$. The activation function used will be marked by the arrow connecting the different layers.

A.6 Autoencoder

The autoencoder is a special type of feed-forward neural network (see Appendix A.5), first proposed by Rumelhart et al. in 1986 [74], with this work being improved upon by Kramer in 1991 [75]. The autoencoder in its most basic form is distinguished by two characteristic properties.

- Firstly, input layer $\boldsymbol{x}^{(0)}$ and output layer $\boldsymbol{x}^{(L)}$ have to have the same dimensionality. In this work, the input layer will be denoted as $\boldsymbol{x} = \boldsymbol{x}^{(0)}$, while the output layer will be $\widehat{\boldsymbol{x}} = \boldsymbol{x}^{(L)}$.
- An autoencoder has at least one hidden layer ($L > 1$) with fewer nodes than input and output layer. The hidden layer l_B with the lowest number of nodes is commonly known as bottleneck layer. The bottleneck layer is named $\boldsymbol{z} = \boldsymbol{x}^{(l_B)} \in \mathbb{R}^m$.

It is possible to split the autoencoder network into two separate networks, the encoder E and the decoder D (in Figure 8a, a method used for visualizing autoencoders can be seen):

$$\begin{aligned}
E_{\boldsymbol{\theta}_E} &= \left(f^{(l_B)}_{\boldsymbol{\theta}^{(l_B)}} \circ \ldots \circ f^{(1)}_{\boldsymbol{\theta}^{(1)}}\right), & \boldsymbol{\theta}_E &= \left\{\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(l_B)}\right\} \\
D_{\boldsymbol{\theta}_D} &= \left(f^{(L)}_{\boldsymbol{\theta}^{(L)}} \circ \ldots \circ f^{(l_B+1)}_{\boldsymbol{\theta}^{(l_B+1)}}\right), & \boldsymbol{\theta}_D &= \left\{\boldsymbol{\theta}^{(l_B+1)}, \ldots, \boldsymbol{\theta}^{(L)}\right\}
\end{aligned} \tag{43}$$

In this work, the domain of input and output layer will be referred to as design space X ($\boldsymbol{x}, \widehat{\boldsymbol{x}} \in X$), while the domain of $\boldsymbol{z}$ will be named latent space Z ($\boldsymbol{z} \in Z$). The following can then be assumed:

$$\widehat{\boldsymbol{x}} = D_{\boldsymbol{\theta}_D}(\boldsymbol{z}) = D_{\boldsymbol{\theta}_D}(E_{\boldsymbol{\theta}_E}(\boldsymbol{x})) \tag{44}$$

The corresponding reconstruction loss function $\mathcal{L}_{R,i}$ for a single sample $\boldsymbol{x}_i$ is then the mean squared error, where $\boldsymbol{w}_i$ is the corresponding weight matrix:

$$\mathcal{L}^{E,D}_{\mathrm{R},i} = \langle \boldsymbol{w}_i, (\boldsymbol{x}_i - D(E(\boldsymbol{x}_i)))^2 \rangle_F \tag{45}$$

The E, D in the power of the loss function indicate which network parameters are optimized depending on this loss, which in this case are the encoder network E and decoder network D.
There can be a differentiation between deep and shallow autoencoders, with shallow autoencoder being characterized by $L = 2$, while on can find $L > 2$ for deep autoencoders [76].

A.7 Autoencoder variations

Using only encoder E and decoder D might be possible, but the literature indicates that expanding the network might be beneficial [45, 46, 48]. Based on these works, four different network combinations are possible (see Figure 8).

- When using a discriminator network D_{Dis}, the expanded network is known as an adversarial autoencoder [77]. The main purpose of its use is to enforce a certain distribution of the encoded training samples in latent space, which would lead to feasible design being produced over the whole latent space [48]. While adversarial autoencoders are similar in that purpose to generative adversarial networks [60], they do not suffer from mode collapse [78] and allow for easier training due to the lower dimensional input of the discriminator[48]. To achieve this, the discriminator maps a latent space representation $\boldsymbol{z}$ onto the likelihood $L = D_{\text{Dis}}(\boldsymbol{z}) \in [0, 1]$ that it has been generated according to the desired distribution $\mathcal{Z}$. To achieve this, the discriminator is trained to differentiate between two different sources for latent space samples, namely random samples ($\boldsymbol{z}_i \sim \mathcal{Z}$) and encoded training set samples ($E(\boldsymbol{x}_i)$ with $\boldsymbol{x}_i \in \boldsymbol{X}_\lambda$). During training, the following loss function is applied, which is used to optimize the parameters $\boldsymbol{\theta}_{D_{\text{Dis}}}$ of the discriminator:

$$\mathcal{L}_{\text{D1},i}^{D_{\text{Dis}}} = \frac{1}{2} \left(\ln \left(D_{\text{Dis}}(E(\boldsymbol{x}_i)) \right) + \ln \left(1 - D_{\text{Dis}}(\boldsymbol{z}_j) \right) \right) \tag{46}$$

  This forces the discriminator to put out numbers close to 0 for samples originating from the training set, while samples generate according to the desired distribution are forced towards $L = 1$.
  On the other hand, the encoder is trained to fool the discriminator, with it trying to minimize the following loss function, so that training samples tend towards $L = 1$:

$$\mathcal{L}_{\text{D2},i}^{E} = \ln \left(1 - D_{\text{Dis}}(E(\boldsymbol{x}_i)) \right) \tag{47}$$

  These two loss functions are each randomly added to the overall loss function of the network for each batch with a likelihood of P_{Dis}, so that none, one or both of the loss functions can be used in each batch. The corresponding network setups using D_{Dis} can be seen in Figures 8b and 8d.

- One can also add a surrogate network S, which maps a latent space vector $\boldsymbol{z}$ onto the cost function approximation $\widehat{C} = S(\boldsymbol{z}) \in [0, 1]$. This could enforce a better separation between better and worse samples in the latent space, possibly reducing the number of local minima for $c(D(\boldsymbol{z}))$, which would allow for a faster optimization [45, 46]. Here, the true value that the networks is trained to assume is

$$\begin{aligned} \widehat{c}_i &= \frac{1}{10} + \frac{8}{10} \frac{c(\boldsymbol{x}_i) - c_{\min}}{c_{\max} - c_{\min}} \\ c_{\min} &= \min_{\boldsymbol{x}_i \in \boldsymbol{X}_\lambda} c(\boldsymbol{x}_i) \\ c_{\max} &= \max_{\boldsymbol{x}_i \in \boldsymbol{X}_\lambda} c(\boldsymbol{x}_i) \end{aligned} \tag{48}$$

  Here, the following loss function is then used to train the encoder E and the surrogate model S:

$$\mathcal{L}_{\text{surr},i}^{E,S} = \beta_S \left(\widehat{c}_i - S(E(\boldsymbol{x}_i)) \right)^2 \tag{49}$$

  This loss function is added to the overall loss function of the whole network. Then, the corresponding networks using S can be seen in Figures 8c and 8d.

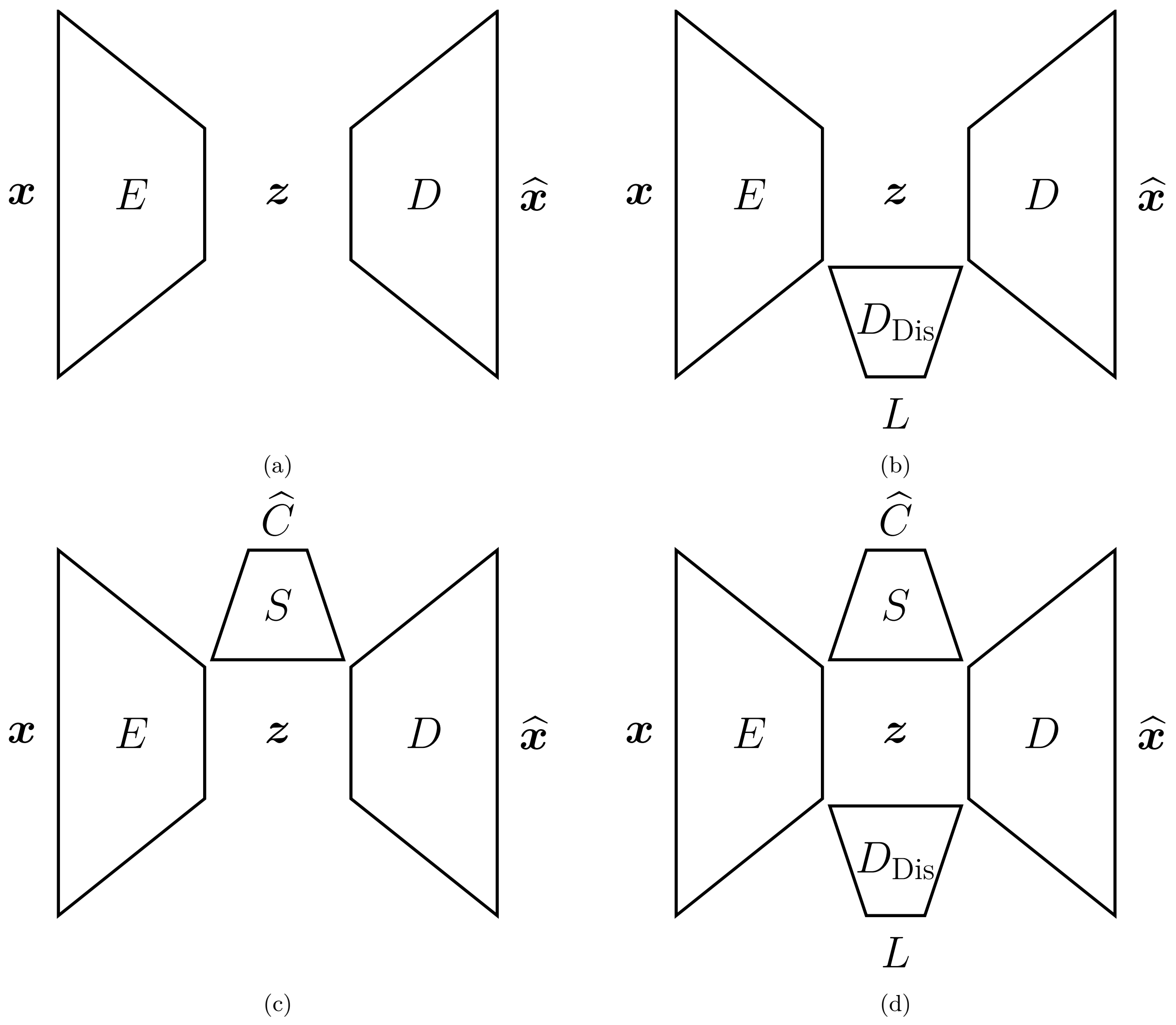


Figure 8: A depiction of different Autoencoder networks with encoder E and decoder D. Discriminator D_{Dis} and surrogate model S can be added to the network.

A.8 Signed Distance Field

The main goal of the autoencoder is to reproduce the input layer at the output layer as closely as possible. In the case of topology optimization problems like compliance minimization (see section 1.4 for an example), the boundaries of the design between material and void are most important. To allow the autoencoder to mainly focus on these parts of the design, a transformation $SDF(\boldsymbol{x}) : [0,1]^{n_y \times n_x} \to \mathbb{R}^{n_y \times n_x}$ to a quasi-signed distance field is created, which according to some papers leads to a better autoencoder performance [79].
The SDF function itself consists out of two steps.

- In the first, one will get $\boldsymbol{x}_S$

$$[x_S]_{k,l} = \begin{cases} -2\sqrt{n_y^2 + n_x^2} & [x]_{k,l} \leq 0.002 \\ [x]_{k,l} - 0.5 & 0.002 < [x]_{k,l} \leq 0.998 \\ 2\sqrt{n_y^2 + n_x^2} & 0.998 < [x]_{k,l} \end{cases} \tag{50}$$

  Here, the term $2\sqrt{n_y^2 + n_x^2}$ is used to indicate which elements still need to get a distance value assigned, as this term cannot occur naturally as a distance in a mesh of the size $n_y \times n_x$.

- In a second step, one will than iterate again and again over the mesh elements, whose absolute value $| [x_S]_{k,l} |$ is still equal to $2\sqrt{n_y^2 + n_x^2}$. These elements $[x_S]_{k,l}$ will be updated by firstly constructing the matrix $\boldsymbol{S}_{k,l}$

$$\boldsymbol{S}_{k,l} = \operatorname{sgn}\left([x_S]_{k,l}\right) \begin{pmatrix} [x_S]_{k-1,l-1} & [x_S]_{k-1,l} & [x_S]_{k-1,l+1} \\ [x_S]_{k,l-1} & [x_S]_{k,l} & [x_S]_{k,l+1} \\ [x_S]_{k+1,l-1} & [x_S]_{k+1,l} & [x_S]_{k+1,l+1} \end{pmatrix} \tag{51}$$

  If an element in $\boldsymbol{S}_{k,l}$ lies outside of $\boldsymbol{x}_S$, the closest value will be taken instead. For example, $[x_S]_{0,1}$ will be replaced with $[x_S]_{1,1}$.
  Meanwhile, a filter matrix $\boldsymbol{\mathcal{F}}$ is defined:

$$\boldsymbol{\mathcal{F}} = \begin{pmatrix} \sqrt{2} & 1 & \sqrt{2} \\ 1 & 0 & 1 \\ \sqrt{2} & 1 & \sqrt{2} \end{pmatrix} \tag{52}$$

  One can then can get the matrix $\widetilde{\boldsymbol{S}}_{k,l}$:

$$\left[\widetilde{S}_{k,l}\right]_{c,d} = \max\left\{ [S_{k,l}]_{c,d}, -\frac{1}{2}[\mathcal{F}]_{c,d} \right\} \tag{53}$$

  One can than overwrite $[x_S]_{k,l}$:

$$[x_S]_{k,l} =: \operatorname{sgn}([x_S]_{k,l}) \min_{c,d} \left\{ \left[\widetilde{S}_{k,l}\right]_{c,d} + [\mathcal{F}]_{c,d} \right\} \tag{54}$$

  After iterating until the maximum absolute value of $\boldsymbol{x}_S$ is smaller than $2\sqrt{n_y^2 + n_x^2}$, the final outcome $\boldsymbol{x}_S = SDF(\boldsymbol{x})$ is achieved.

To transform signed distance fields back to density fields, one can use the function $SDF^{-1} : \mathbb{R}^{n_y \times n_x} \rightarrow [0,1]^{n_y \times n_x}$ with $SDF^{-1}(SDF(\boldsymbol{x})) = \boldsymbol{x}$

$$\left[SDF^{-1}(\boldsymbol{x})\right]_{k,l} = \begin{cases} 0 & [x]_{k,l} \leq -0.5 \\ [x]_{k,l} + 0.5 & -0.5 < [x]_{k,l} \leq 0.5 \\ 1 & 0.5 < [x]_{k,l} \end{cases} \tag{55}$$

It has to be noted that consequently in the loss functions from equations (45), (46), (47), (49), and (60), $\boldsymbol{x}_i$ has to be replaced with $SDF(\boldsymbol{x}_i)$, if the signed distance field has to be used.

A.9 Neural Network training

To train the neural networks in this work, backpropagation and gradient based optimization is used. This makes the performance of the trained networks dependent on the initial parameters set in the networks. Bias vectors were initialized as zero, while weight tensors $\boldsymbol{W}^{(l)}$ and filters $\boldsymbol{K}^{(l)}$ are set using Xavier initialization [80]. In that, they are generated according to a normal distribution with mean of zero and a standard deviation of

$$\sigma_W^{(l)} = \sqrt{\frac{2}{\nu^{(l)} + \nu^{(l-1)}}} \tag{56}$$

and

$$\sigma_K^{(l)} = \sqrt{\frac{2}{t_y^{(l)} t_x^{(l)} \left(\kappa^{(l)} + \kappa^{(l-1)}\right)}}. \tag{57}$$

After initializing the network parameters $\boldsymbol{\theta}_0$, batch gradient descent is used to optimize the network parameters over multiple epochs. In each epoch, the training samples are being randomly sorted into n_b batches for each epoch. The loss function for each batch $\mathcal{L}$ is than calculated by averaging over the loss $\mathcal{L}_i$ of every sample in the batch, and the corresponding gradient of the loss over the network parameters $\boldsymbol{g}$ is calculated using backpropagation. This is done for every batch in every epoch.
Adam [55] (see Appendix A.1) is then used to update the network parameters after every batch from $\boldsymbol{\theta}_i$ to $\boldsymbol{\theta}_{i+1}$. In this work, a learning rate of $\alpha = 0.001$, as well as decay rates $\beta_1 = 0.9$ and $\beta_2 = 0.999$ are used when training a neural network.

A.10 Pretraining

For autoencoder, there is always the possibility of using pretraining for the encoder E and decoder D, as this might improve the autoencoder performance, although there is conflicting evidence [81, 82].
For pretraining to be possible, there would need to be several corresponding layers with the same dimensionality in both encoder E and decoder D. This would than allow one to split up both into n_{pre} smaller networks E_i and D_i, where the input layer of E_i and the output layer of D_i would have the same dimension, as well as the output layer of E_i and the input layer of D_i:

$$\begin{aligned} E &= \left(E_{n_{\text{pre}}} \circ \ldots \circ E_1\right) \\ D &= \left(D_1 \circ \ldots \circ D_{n_{\text{pre}}}\right) \end{aligned} \tag{58}$$

One could then train these smaller networks, beginning with E_1 and D_1 and then going inwards from there, by minimizing either the loss function

$$\mathcal{L}_{\text{pre1},i}^{E_j,D_j} = \overline{\left(\boldsymbol{x}_i - \left(D_1 \circ \ldots \circ D_j \circ E_j \circ \ldots \circ E_1\right)\left(\boldsymbol{x}_i\right)\right)^2} \tag{59}$$

or

$$\mathcal{L}_{\text{pre2},i}^{E_j,D_j} = \overline{\left(\left(E_{i-1} \circ \ldots \circ E_1\right)\left(\boldsymbol{x}_i\right) - \left(D_j \circ E_j \circ \ldots \circ E_1\right)\left(\boldsymbol{x}_i\right)\right)^2}. \tag{60}$$

While the first loss function overall leads to smaller losses, this is achieved by a longer calculation time. Therefore, in this work, $\mathcal{L}_{\text{pre2}}$ is used for pretraining.

B On the existence of X'

As it is technically true that one can find a one-dimensional X' which includes all local minima of any arbitrary cost function by simply connecting them one after another by straight lines, another important point has to be added to the assumption used in this work. This is that X' can be recovered when only using a subset of all the local minima.

For instance, if one had all local minima on a plane in a higher dimensional space, one could connect them all in a line. But this line will vary if one only uses some of the local minima to construct it, and therefore, it is unlikely that the global optimum will be part of this line if it was not used to construct it. On the other hand, one would only need to use 3 random local minima to construct a plane which in every case will include the global optimum. Consequently, one would consider that X' had a dimensionality of $m = 2$. More, but again likely not all local minima would be required to construct X' if this manifold was curved.

Mathematically, this could be described by assuming that there exists a transformation $T_{\boldsymbol{Y}}$, parameterized by a proper subset $\boldsymbol{Y}$ of the set of all local minima $\boldsymbol{X}$:

$$\exists\, T_{\boldsymbol{Y}} : \mathbb{R}^m \rightarrow \mathbb{R}^n \left(\forall \boldsymbol{x} \in \boldsymbol{X}\ \exists\, \boldsymbol{z}\ \big(\boldsymbol{x} = T(\boldsymbol{z})\big)\right), \quad \boldsymbol{Y} \subset \boldsymbol{X} = \{\boldsymbol{x} | \nabla c(\boldsymbol{x}) = \boldsymbol{0}\} \tag{61}$$

C Calculate probability of good results in the design domain

To find samples belonging to X', local optimization is proposed. But one could question, if other methods like random sampling might also be used to find designs with a low cost function, who likely are close to a local minimum.
For an examples, one will use compliance minimization on a 40×100 mesh, and the typical bridge problem (see Figure 18). When randomly sampling 400.000 points, one can find that the minimum compliance value is $C_{\min} \approx 1792.3$. Consequently, the event $C < C_{min}$ did not happen once in 400.000 tries.
From this, one could try to approximate the probability $p = P\left(C < C_{\min}\right)$ [83]. In a first step, one sets the confidence for this approximation, which in this case will be chosen to be 95%. Then, one can calculate the confidence interval of $p \in [p1, p2]$, also called Wilson interval:

$$\left(\frac{f}{n} - p_{1,2}\right)^2 = k^2 \frac{p_{1,2}\left(1 - p_{1,2}\right)}{n} \tag{62}$$

The selected confidence of 95% will lead to $k = 1.96$ in equation (62), with $f = 0$ being the number of times the event $C < C_{\min}$ has happened and $n = 400000$, leading to the confidence interval $p \in [0, 9.604e-6]$.
By the use of local optimization, it can be shown that there are designs which allow for a compliance of below $C < 200$ for the given problem, with the probability $P(C < 200)$ consequently even smaller than p. Therefore, the probability of finding samples which have a low compliance, and therefore likely belong to X', using a random search is vansihingly low.

D The continuity of the decoded latent space X_Z in the design space X.

Firstly, it has to be noted that both Encoder E and Decoder D are continuous function (this is a necessary condition for them to be differentiable, which is a requirement for backpropagation-based network training). Meanwhile, in this work, it can be assumed that $Z = [0,1]^m$, with $E : X \rightarrow Z$, $D : Z \rightarrow X_Z$ and $X_Z \subset X$.
Under these conditions, it can be assumed that X_Z will be an m-dimensional, continuous manifold, as the Decoder just deforms Z, with neighboring points staying closely together.

$$D(\boldsymbol{z} + d\boldsymbol{z}) = D(\boldsymbol{z}) + \boldsymbol{\nabla} D(\boldsymbol{z}) d\boldsymbol{z}. \tag{63}$$

Consequently, there can also be no holes in X_Z, as some points on opposite sides of the hole would have need to be far apart, although they were close together in Z (which had no holes, which would contradict equation (63)

E Explaining results from section 1.3

The tendency of autoencoders encoding evenly spread samples $\boldsymbol{X}$ from a n-dimensional manifold onto a m-dimensional latent space Z to include the mean of the samples $\overline{\boldsymbol{x}}$ in the decoded latent space X_Z has been observed.
This can likely be explained by the fact that the mean has the lowest reconstruction error:

$$\overline{\boldsymbol{x}} = \underset{\boldsymbol{x}\in X}{\operatorname{argmin}} \sum_i \|\boldsymbol{x}_i - \boldsymbol{x}\|^2 \tag{64}$$

This can be proven by firstly considering a one-dimensional problem:

$$\begin{aligned}\sum_i (x_i - x)^2 &= \sum_i \left(x_i^2 - 2x_i x + x^2\right) \\ &= \sum_i \left(x_i^2\right) - 2x \sum_i (x_i) + nx^2 \\ &= \sum_i \left(x_i^2\right) - n\overline{x}^2 + n\overline{x}^2 - 2nx\overline{x} + nx^2 \\ &= \sum_i \left(x_i^2\right) - n\overline{x}^2 + n(x - \overline{x})^2 \\ &= n(x-\overline{x})^2 + c\end{aligned} \tag{65}$$

$$\underset{x}{\operatorname{argmin}} \sum_i (x_i - x)^2 = \underset{x}{\operatorname{argmin}}\, n(x-\overline{x})^2 + c = \overline{x} \tag{66}$$

As $\sum_i ||\boldsymbol{x}_i - \boldsymbol{x}||^2$ is just the sum of $\sum_i (x_{i,j} - x_j)^2$ over each dimension j, the minimum of the former has to be minimal if each dimension of $\boldsymbol{x}$ takes the average value of the samples in this dimension. Therefore, the minimum of the former equation can be found at $\boldsymbol{x} = \overline{\boldsymbol{x}}$.

F Proof of concept 1

A function c is designed in multiple steps to test the viability of the proposed method.

1. In a $M = 10$ dimensional space ($Y = [0,1]^M$), 2500 random points $\boldsymbol{y}_i \in Y$ are generated. These points $\boldsymbol{y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{2500}\}$ are then saved.

2. A mapping $T_D : \mathbb{R}^M \rightarrow \mathbb{R}^n$ is created, which allows to find $\boldsymbol{x}_i \in \mathbb{R}^n$ from $\boldsymbol{y}_i$ with $n = 1000$:

$$\begin{aligned}
\boldsymbol{x}_{1,i} =& \tanh\left(\boldsymbol{W}_1 \boldsymbol{y}_i + \boldsymbol{b}_1\right) \\
\boldsymbol{x}_{2,i} =& \exp\left(-\left(\boldsymbol{W}_2 \boldsymbol{x}_{1,i} + \boldsymbol{b}_2\right) \odot \left(\boldsymbol{W}_2 \boldsymbol{x}_{1,i} + \boldsymbol{b}_2\right)\right) \\
\left[x_{3,i}\right]_j =& \frac{2}{1 - \left[b_3\right]_j} \left(\left[x_{2,i}\right]_j - \left[b_3\right]_j\right) - 1 \\
\boldsymbol{x}_{4,i} =& \boldsymbol{W}_4 \boldsymbol{x}_{3,i} + \boldsymbol{b}_4 \\
\boldsymbol{x}_{5,i} =& \frac{4}{5} \tanh\left(\boldsymbol{x}_{4,i} - \boldsymbol{x}_{4,1}\right) \\
\left[x_i\right]_j =& \left(\frac{3}{4}\left[x_{5,i}\right]_j - \frac{1}{4}\right)\left(\left[x_{5,i}\right]_j^2 - 1\right) + \frac{2}{\pi} \arcsin\left(\left[x_{5,i}\right]_j\right) \\
& \boldsymbol{W}_1 \in \mathbb{R}^{n \times M},\ \left[W_1\right]_{k,l} \sim U\left(\frac{1}{20}, \frac{1}{10}\right) \operatorname{sgn}\left(U(-1,1)\right) \\
& \boldsymbol{W}_2 \in \mathbb{R}^{n \times n},\ \left[W_1\right]_{k,l} \sim U\left(\frac{1}{20}, \frac{1}{10}\right) \operatorname{sgn}\left(U(-1,1)\right) \\
& \boldsymbol{W}_4 \in \mathbb{R}^{n \times n},\ \left[W_1\right]_{k,l} \sim U\left(\frac{1}{20}, \frac{1}{10}\right) \operatorname{sgn}\left(U(-1,1)\right) \\
& \boldsymbol{b}_1 \in \mathbb{R}^{n},\ \left[b_1\right]_j \sim U\left(-\frac{1}{2}, \frac{1}{2}\right) \\
& \boldsymbol{b}_2 \in \mathbb{R}^{n},\ \left[b_1\right]_j \sim U\left(-\frac{1}{2}, \frac{1}{2}\right) \\
& \left[b_3\right]_j = \min_i \left[x_{2,i}\right]_j \\
& \boldsymbol{b}_4 \in \mathbb{R}^{n},\ \left[b_1\right]_j \sim U\left(-\frac{1}{2}, \frac{1}{2}\right)
\end{aligned} \tag{67}$$

This leads to $\boldsymbol{X} = T_D(\boldsymbol{Y}) = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_{2500}\}$, where $\boldsymbol{x}_1 = \frac{1}{4}\mathbf{1}$, with $\mathbf{1} \in \mathbb{R}^n$ having a value of 1 for every element. These points will be part of an continuous m-dimensional manifold in $\mathbb{R}^n$.

3. The function c_1 can than be build:

$$c_1(\boldsymbol{x}) = \min_i ||\boldsymbol{x} - \boldsymbol{x}_i||^2 \tag{68}$$

This function will have multiple local minima, with $c_1(\boldsymbol{x}_i) = 0$.

4. The function c_2 can be build:

$$c_2(\boldsymbol{x}) = \left(1 - \exp\left(-10 \left\|\boldsymbol{x} - \boldsymbol{x}_1\right\|^2\right)\right)\left(\frac{2}{5} + \exp\left(-10 \left\|\boldsymbol{x} - \boldsymbol{x}_1\right\|^2\right)\right) \tag{69}$$

This function will have one global optimum at $\boldsymbol{x}_1$. Up until a distance of $R = 0.385$ from this optimum, local optimization should lead to the global optimum, while points farther away should not be able to get to the global optimum using local optimization.

5. The final function c can than be found, with the global minimum at $\boldsymbol{x}_1$:

$$c(\boldsymbol{x}) = c_1(\boldsymbol{x}) + c_2(\boldsymbol{x}) \tag{70}$$

This function is then optimized in two different ways:

- Firstly, the proposed method from 1.2 is used, with the four steps being implemented as follows:
  1. 10000 random samples are created by sampling the design space $X = [-1, 1]^n$ uniformly. From these points, the training set $\boldsymbol{X}_{500}$ is created using Adam with $\alpha = 0.01$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$.

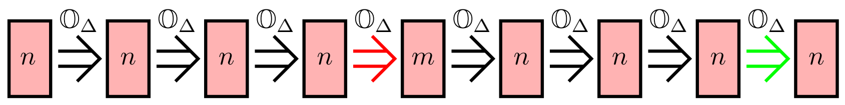


Figure 9: Black connection use the activation function σ_1, red ones use σ_2, and green ones use σ_3 (see equation (74)).

2. A neural network (see Figure 9) is than trained with mean squared error loss function for 250 epochs, with the first 225 epochs having 10 batches each, while the last 25 epochs only use one batch. The latent space dimension m is chosen as $m = M = 10$, as well as $m = 5$ for a second case.
3. The optimization over latent space is than performed, using the cost function c_μ (see equation (3)) with $\mu = 5$. Here, on uses differential evolution $DE_{50,500,0.6,0.9}$ for the optimization of c_5, and Adam with $\alpha = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ for the local optimization steps LO inside of c_5
4. Post-processing consists of $\nu = 1000$ steps, using Adam with $\alpha = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$, and uses the from from equation (6).

- Alternatively, an optimization over the whole latent space is performed using differential evolution and post processing. Here, during the first steps, $DE_{1000,5000,0.6,0.9}$ is used, while in the post-processing, 500 steps of Adam with $\alpha = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ are performed. This result is not considered in section 1.3.

Results

In Figure 10, it can be seen that the proposed method is able to find the global optimum of c, something that a more standard approach of differential evolution in the design space was not able to do in the same timeframe. This is caused as the global optimum is likely included in the decoded latent space X_Z. An example for such a case can be seen in Figure 15. But it has also to be mentioned that even if there exists a M dimensional manifold X' on which all local optima lie, than this does not guarantee that the decoded latent space X_Z will be able to reproduce this behavior (see Figure 16)

G Proof of concept 2

Four benchmark functions from a paper by Abualigah et al. [62] will be optimized using the method proposed in section 1.2.

$$
\begin{aligned}
f_1(\boldsymbol{x}) =& \sum_{i=1}^{n} \left(-x_i \sin\left(\sqrt{|x_i|}\right)\right) + 418.9829n, && \boldsymbol{x} \in [-500, 500]^n \\
f_2(\boldsymbol{x}) =& 20\left(1 - \exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right)\right) + \exp(1) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right), && \boldsymbol{x} \in [-32, 32]^n \\
f_3(\boldsymbol{x}) =& (\boldsymbol{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n}(x_i^2) - \prod_{i=1}^{n}\left(\cos\left(\frac{x_i}{\sqrt{i}}\right)\right), && \boldsymbol{x} \in [-500, 500]^n \\
f_4(\boldsymbol{x}) =& \frac{\pi}{n}\left(10\sin(\pi y_1)^2 + \sum_{i=1}^{n-1}(y_i - 1)^2\left(1 + 10\sin(\pi y_{i+1})^2 + u\right)\right), && \boldsymbol{x} \in [-50, 50]^n \\
& y_i = \frac{x_i + 5}{4}, \quad u = \sum_{i=1}^{n} 100\max\{0, |x_i| - 10\}^4
\end{aligned}
\tag{71}
$$

In each case, the original dimension is chosen to be $n = 100$, with the latent space having the dimension $m = 5$. For all these functions, the minimal value is $f_i = 0$.
For each step, the following exact methods have been used:

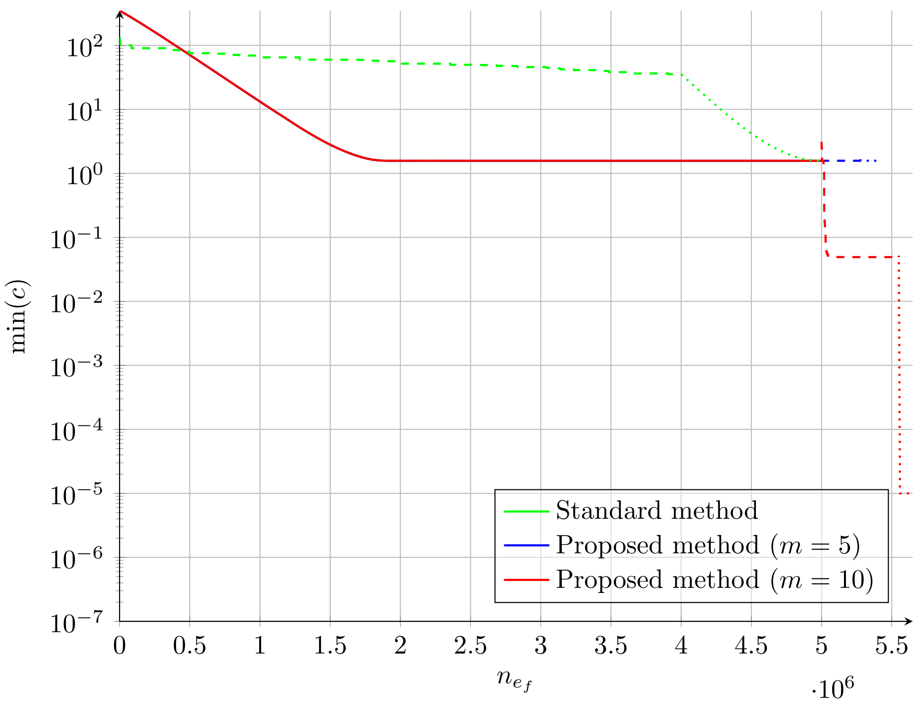


Figure 10: The results of optimizing c. The continuous lines describe the first step, where λ steps of local optimization are used on 10000 random initial points. The dashed line shows the minimum during the optimization over latent space, while the dotted line shows the best compliance during post-processing.

- The training set is generated by optimizing $N = 10000$ random samples using Adam (parameters in Table 2). But only the half of the found samples with the lower function values is used for training the neural network, firstly to speed up training and secondly to avoid training on samples with overall worse fitness. In section 1.3, one only uses the case of $\lambda = 125$, but the cases $\lambda = 5$ and $\lambda = 25$ will also be considered in this part of the appendix

Table 2: Parameters of Adam used for local optimization.

Function	Learning rate α	Decay rate β_1	Decay rate β_2
f_1	20	0.9	0.999
f_2	1	0.9	0.999
f_3	30	0.9	0.999
f_4	2.5	0.5	0.75

- The samples are firstly normalized, with:

$$\boldsymbol{X}_{\text{train}} = \frac{1}{x_{\max}} \boldsymbol{X}_{\lambda}. \tag{72}$$

This is done as the Neural Neutwork output is limited to the domain $[-1, 1]^n$, while $\boldsymbol{x} \in [-x_{\max}, x_{\max}]^n$. It also implies that the decoded samples $D(\boldsymbol{z})$ will have to be multiplied with the factor $x_{\max}$ before evaluating the function value.
Here, the autoencoder (see Figure 11) is trained using Adam for 100 epochs, with 20 batches each, with the loss function being the mean squared error.

- For optimization in latent space, differential evolution $DE_{5m,2000,0.6,0.9}$ has been used to optimize the cost function C_0.

- Finally, during the post-processing, Adam was used for $\nu = 2000$ iterations with parameters from Table 3.

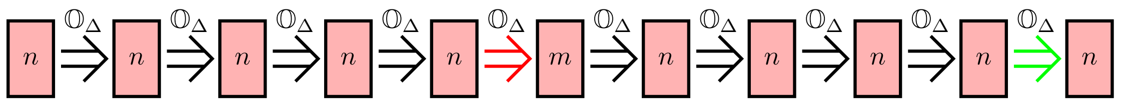


Figure 11: Black connection use the activation function σ_1, red ones use σ_2, and green ones use σ_3 (see equation (74)).

Table 3: Parameters of Adam used for post-processing.

Function	Learning rate α	Decay rate β_1	Decay rate β_2
f_1	0.5	0.9	0.999
f_2	0.00025	0.9	0.999
f_3	0.05	0.9	0.999
f_4	0.05	0.9	0.999

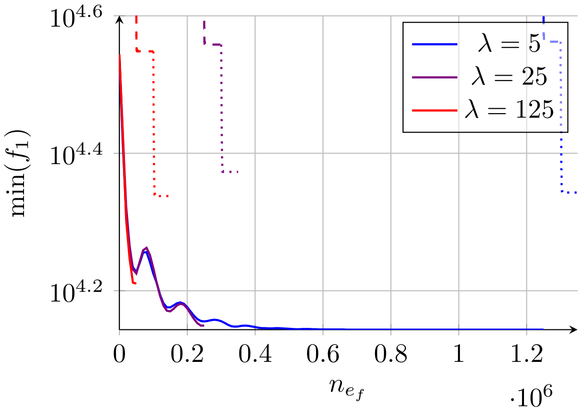
Results

When comparing the different results in Figure 12, it is clear that in cases of f_1 and f_2, one was unable to find the minimum value of $\min(f) = 0$, while this was possible for all values of λ in f_3 and f_4.
In regard to the fact that no global optimum was found for f_1, it can be argued that this is the case as f_1 does not fulfill the assumption made in constructing the proposed method, as its local optima are nearly evenly spread around the domain X. Therefore, X' is similar to X and can not accurately be recreated by X_Z, which is produced by the autoencoder.
Meanwhile, the proposed method can find the global optimum of f_3 and f_4 up to machine precision, although both cases share the fact with f_1 (and f_2) that the local minima are distributed evenly throughout the whole domain. But they have the main difference that their global optimum, as the one of f_2, lies in the center of the nearly uniformly distributed cluster of training samples on which the autoencoder is trained. And as the autoencoder in such cases then often includes the mean of the training samples (or goes close by, see Appendix E and Figure 17), one can expect the optimum of the optimization over X_Z to be close to the global optimum. This behavior seems to be independent from the dimensionality m of the latent space Z, as the f_3 and f_4 can also be optimized successfully for $m = 2$ (see Figures 13b and 13c).
It has now to be explained why for f_2, no matter the latent space dimensionality m, the global optimum can not be found (see Figures 12b and 13a), despite the similarity to the previous cases. The cause for this can likely be found in the frequency of local optima around the global optimum (see Table 4).

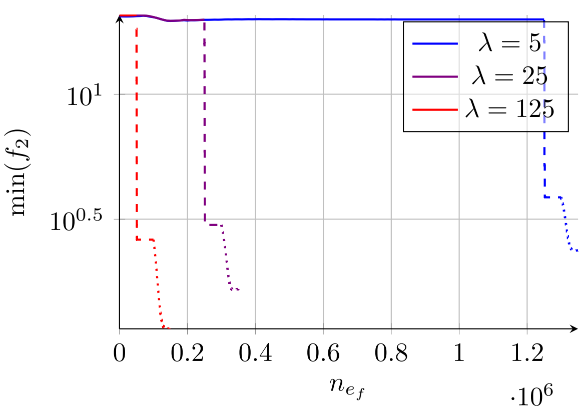
Table 4: Normalized frequency of local minima per dimension k_i and for all dimensions k for different functions. The values of T_i are approximation and only valid near the global optimum.

Property	f_2	f_3	f_4
$x_{\max}$	32	500	50
Period T_i	1	$2\pi\sqrt{i}$	4
$k_i = \frac{T_i}{x_{\max}}$	$\frac{1}{32}$	$\frac{\pi\sqrt{i}}{250}$	$\frac{2}{25}$
$k = \prod_{i=1}^{100} k_i$	$3.0 \cdot 10^{-151}$	$8.1 \cdot 10^{-112}$	$2.0 \cdot 10^{-110}$

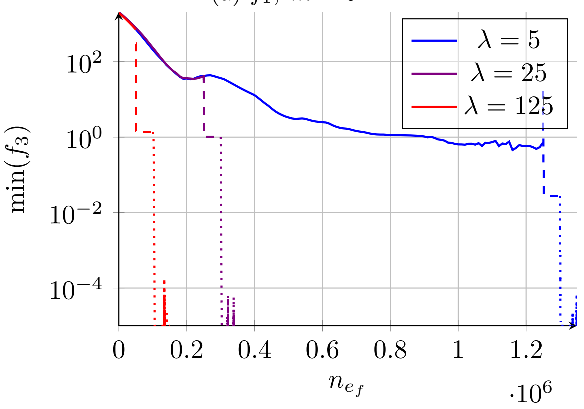
The normalized frequency of local minima is much lower in the case of f_2. Consequently, the part of the domain from which local optimization would lead to a specific optimum is much smaller. Therefore, the decoded latent space X_Z missing the global optimum by a small margin (either due to discrepancies between $\overline{\boldsymbol{x}}$ and the global optimum, or due to the autoencoder not exactly including $\overline{\boldsymbol{x}}$, with both cases visible in Figure 17) is a much greater problem for f_2 than for f_3 and f_4. Consequently, the global optimum cannot be found for f_2 in this case. This explanation is reinforced by Figure 13d, where a reduced domain of f_2 ($\boldsymbol{x} \in [-12.5, 12.5]^n$, which would lead to a similar frequency of local minima as for f_4) allows the proposed method to reliably find the global optimum.



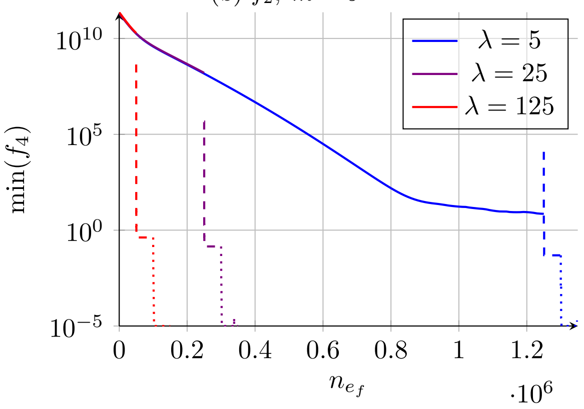
(a) $f_1,\ m = 5$



(b) $f_2,\ m = 5$



(c) $f_3,\ m = 5$



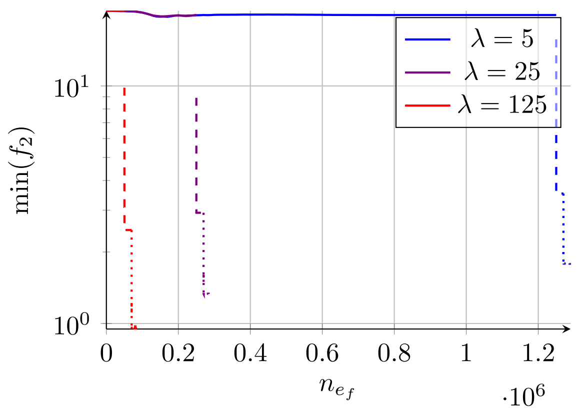
(d) $f_4,\ m = 5$

Figure 12: The results of optimizing the benchmark functions f_i, showing after a number of function evaluations n_{e_f} the corresponding minimum function value $\min(f)$ found. The continuous lines describe the first step, where λ iterations of local optimization (Adam[55]) are used on $N = 10000$ random initial points. The dashed line shows the minimum during the optimization over latent space, while the dotted line shows the best compliance during post-processing.

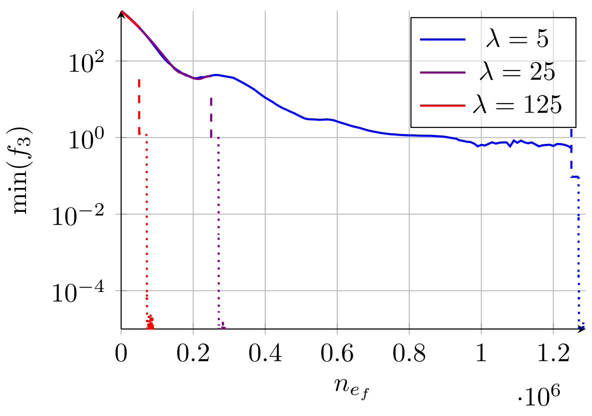
H Examples for different distributions of training samples

Three examples should be given to showcase the possible behavior of the proposed method for global optimization, depending on the distribution of the training samples of the autoencoder. For each of these functions, the design domain is $X = [-1, 1]^2$, with the latent space being $Z = [0, 1]$. Therefore, the autoencoder encodes a $n = 2$ dimensional system into a $m = 1$ dimensional one.
The cost functions which shall be optimized are the following, with $H(x)$ being the Heaviside function:

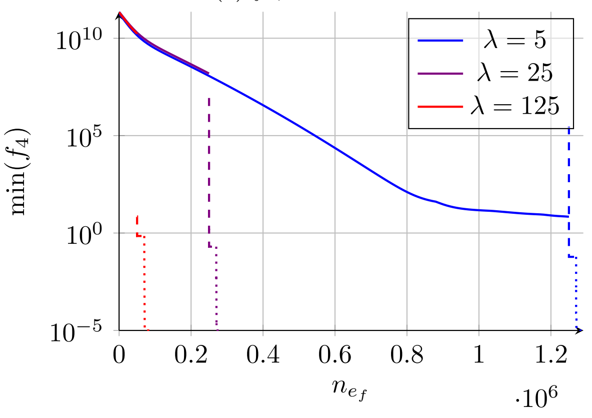
$$\begin{aligned} c_1(x_1, x_2) =& 3\left(1 - \exp\left(-5\left(x_1^4 + 2H(x_1)x_1^2\left(x_2^2 - 0.5625\right) + \left(x_2^2 - 0.5625\right)^2\right)\right)\right) + 1 - \exp\left(-10x_2^4\right) \\ &+ \frac{1}{4}\left(1 - \cos\left(40 \arg(x_1 + ix_2)\right)\right)\left(1 + \cos\left(\arg(x_1 + ix_2)\right)\right)\left(1 - \exp\left(-x_1^2 - x_2^2\right)\right) \\ c_2(x_1, x_2) =& \left(1 - \cos\left(\arg(x_1 + ix_2) - 4\pi\left(x_1^{10} + x_2^{10}\right)^{\frac{1}{5}}\right)\right)\left(1 - \exp\left(-5\left(x_1^{10} + x_2^{10}\right)^{\frac{1}{5}}\right)\right) \\ &+ \frac{1}{5}\left(1 - \exp\left(-5\left((x_1 + 0.5)^2 + x_2^2\right)\right)\right) + \frac{1}{5}\left(1 - \cos\left(40 \arg(x_1 + ix_2)\right)\right)\left(1 - \exp\left(-5\left(x_1^{10} + x_2^{10}\right)^{\frac{1}{5}}\right)\right) \\ c_3(x_1, x_2) =& 5\left(1 - \exp\left(-\sqrt{5\left(x_1^2 + x_2^2\right)}\right)\right) + \exp(1) - \exp\left(\frac{1}{2}\left(\cos(10\pi x_1) + \cos(10\pi x_2)\right)\right) \end{aligned} \tag{73}$$



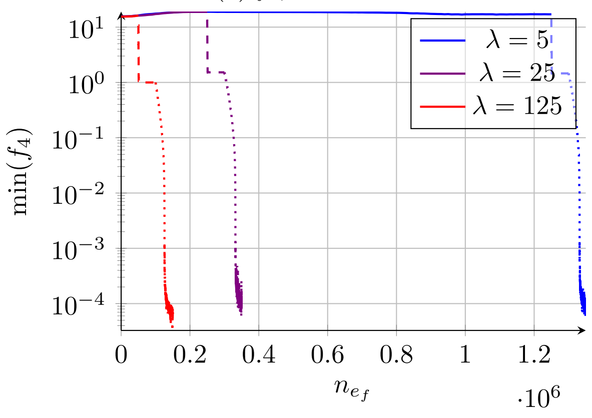
(a) $f_2,\ m = 2$



(b) $f_3,\ m = 2$



(c) $f_4,\ m = 2$



(d) $f_2,\ m = 5$ with reduced domain

Figure 13: The results of f_2, f_3, and f_4, with $n = 100$ and $m = 2$. In Figure 13d, the results for $m = 5$ of f_2 is shown, but this time with a smaller domain for x_i.

For the first step of this process, namely local optimization, Adam [55] is used for $\lambda = 60$ steps, on $N = 250$ samples with the following parameters:

Table 5: Parameters of Adam used for local optimization.

Function	Learning rate α	Decay rate β_1	Decay rate β_2
c_1	0.05	0.9	0.999
c_2	0.02	0.9	0.999
c_3	0.01	0.9	0.999

For the second step, a seven layered deep, dense autoencoder is trained (see Figure 14), for 300 epochs and a batch size of 25, using Adam. In this neural network, three different activation functions are used:

$$\begin{aligned}\sigma_1(x) =& LR(\tanh(x), 0.25)\\ \sigma_2(x) =& \frac{1}{1+\exp(-x)}\\ \sigma_1(x) =& \tanh(x)\end{aligned} \tag{74}$$

Here, LR is the so called Leaky Rectified Linear Unit (LeakyReLU) [84] function:

$$LR(x, a) = \begin{cases} x & x > 0 \\ ax & x \leq 0 \end{cases} \tag{75}$$

The results for the first two steps of the proposed optimization process (see section 1.2), the generation of training samples and the autoencoding, can be seen in Figure 15 for c_1, Figure 16 for c_2, and Figure 17 for c_3.

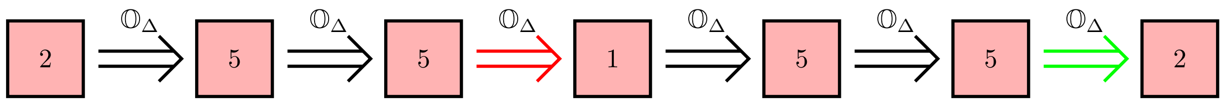


Figure 14: Autoencoder structure used in Appendix H. Black connections use the activation function σ_1, red ones use σ_2, and green ones use σ_3 (see equation (74)). One can find $\hat{\boldsymbol{x}} = D(z) = D(E(\boldsymbol{x}))$

I Implementation of the problem in section 1.4

In this part, the exact implementation of the four steps of the proposed method (see section 1.2) for the compliance minimization problem from section 1.4 is presented.

I.1 Generating the training set

When generating the training set, it is imperative to ensure a lot a variety in between different samples. And while simply using SIMP to advance to each sample from a random initial distribution is possible, this approach is not without flaws. Mainly, it is possible that multiple different starting positions might results in the same result, which would be a waste of time. To avoid this, a method called deflation can be used [58]. In this method, $\boldsymbol{G}_i$ during SIMP (see Appendix A.2) is updated in the following way, which pushes new samples away from already explored designs:

$$\boldsymbol{G}_i =: \boldsymbol{G}_i - 50 \sum_j \frac{\boldsymbol{x}_i - \widetilde{\boldsymbol{x}}_j}{\langle (\boldsymbol{x}_i - \widetilde{\boldsymbol{x}}_j), (\boldsymbol{x}_i - \widetilde{\boldsymbol{x}}_j) \rangle_F^2} \tag{76}$$

The samples are then generated according to the following method:

- 50 initial designs are constructed. Here, the first design $\boldsymbol{x}_{1,0}$ is the homogeneous material distribution, while the other ones $\boldsymbol{x}_{j,0}$ are generated randomly:

$$[\boldsymbol{x}_{j,0}]_{k,l} = U(0,1)^{\frac{3}{2}} \tag{77}$$

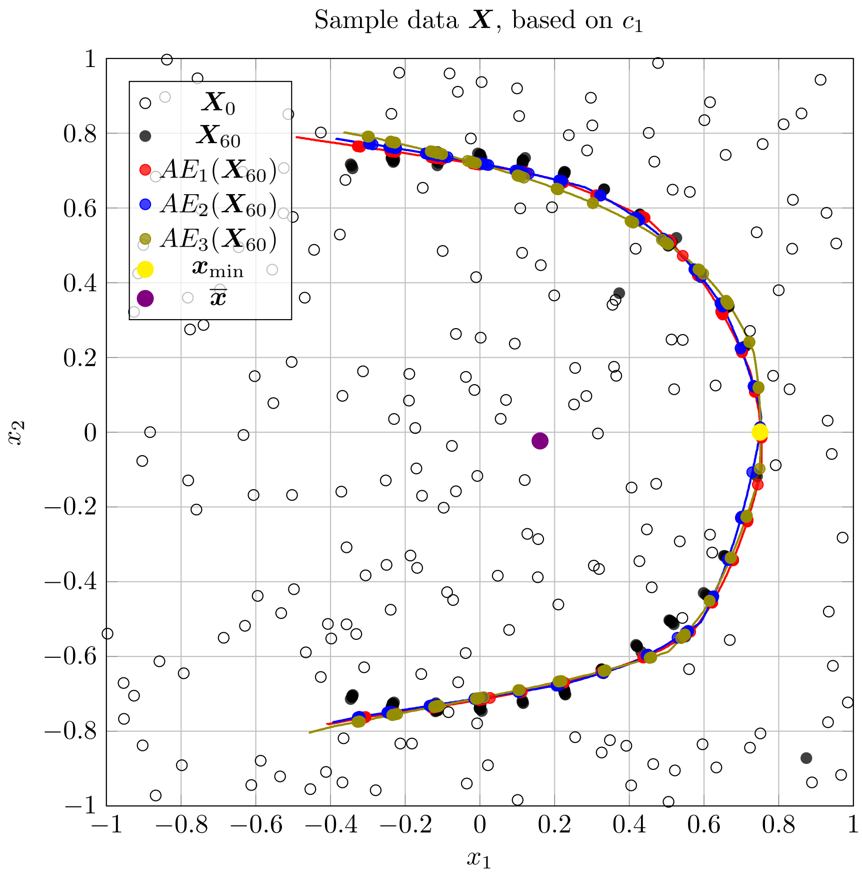
- From each initial distribution $\boldsymbol{x}_{j,0}$, 100 designs are created, one after another. The first design is created without any influence of previous designs, meaning that $\widetilde{\boldsymbol{X}}$ (see equation (76)) is empty. But afterwards, all the designs already produced will be included in $\widetilde{\boldsymbol{X}}$. Two such sets are used here, namely $\widetilde{\boldsymbol{X}}_{25}$ and $\widetilde{\boldsymbol{X}}_{250}$, which include previous designs after 25 and 250 steps of SIMP respectively. $\widetilde{\boldsymbol{X}}_{25}$ is used as testing showed that using two separate steps of deflation would lead to more variety and better results than a one-step approach.
  Each design is then generated as follows, where $\boldsymbol{x}_{0,j}$ is one of the initial distributions, and local optimization is performed using SIMP ($LO(\boldsymbol{x}_i) = SIMP_{\epsilon,\widetilde{\boldsymbol{X}}}(\boldsymbol{x}_i)$, where ϵ is a parameter for the Ginzburg-Landau energy sensitivity (see Appendix A.2), which is used in this case instead of filtering):

$$\boldsymbol{x}_{250,j} = \left(SIMP^{10}_{0.25,\{\}} \circ SIMP^{50}_{1,\{\}} \circ SIMP^{225}_{1,\widetilde{\boldsymbol{X}}_{250}} \circ SIMP^{25}_{1,\widetilde{\boldsymbol{X}}_{25}} \right) (\boldsymbol{x}_{0,j}) \tag{78}$$

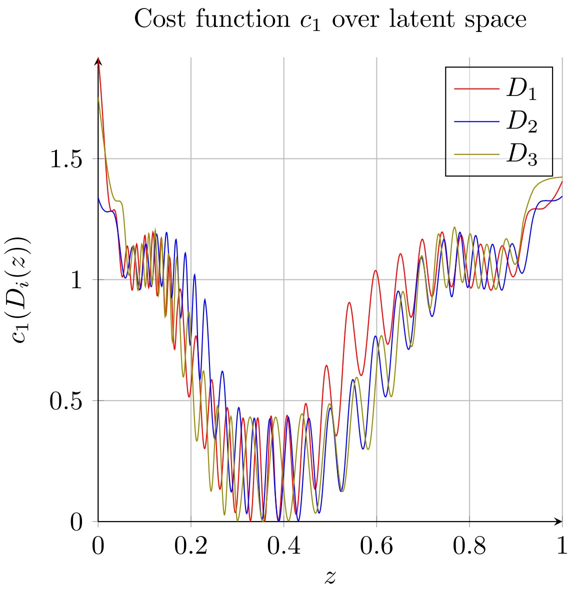
This results in overall 5000 training samples, from which 4000 are randomly selected to train the autoencoder, while the other 1000 are used for validation of the autoencoder training.

I.2 Training the autoencoder

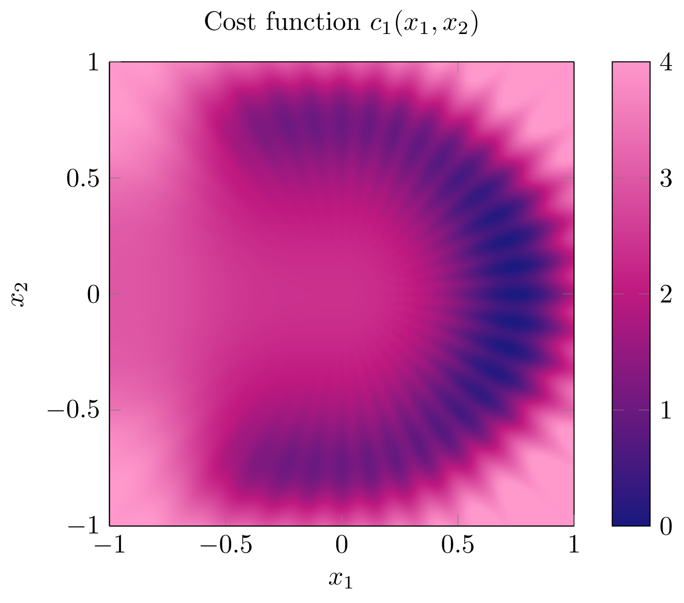
After generating the training set, the samples from it are used to train an adversarial autoencoder (the discriminator network enforces a uniform distribution over the latent space, with $P_{\text{Dis}} = \frac{1}{4}$) with a surrogate model network S (with the weight $\beta_S = \frac{1}{4}$), more information about which can be seen in Figure 8d and Appendices A.6 and A.7. The specific network types used can be seen in Figure 40, and the layer architecture used is mainly chosen based on the results of prior experiments (see Appendix K.2). Additionally, a signed distance field is used to encode the training samples (see Appendix A.8) with the



(a)

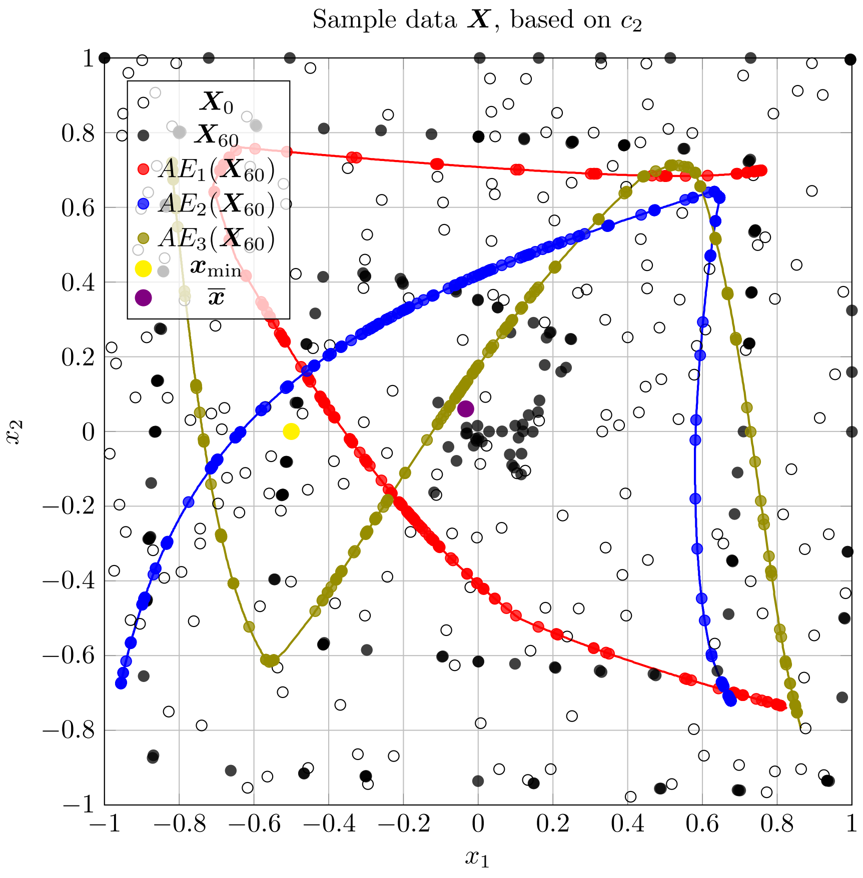


(b)

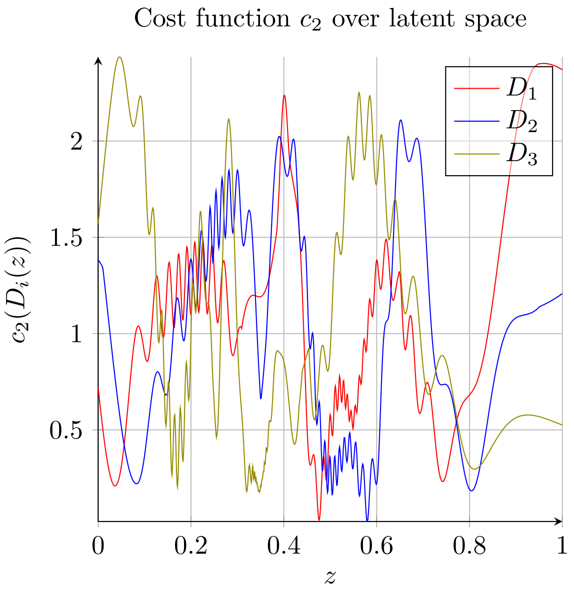


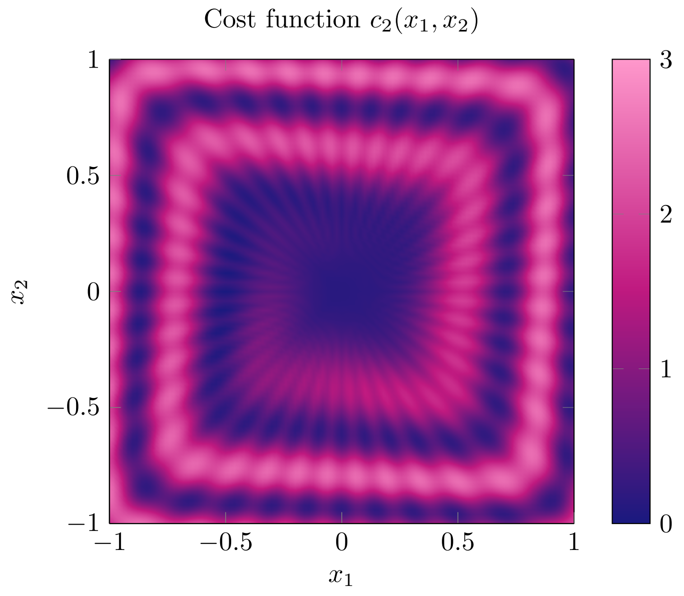
(c)

Figure 15: In part (a) of this figure, one can see the initial random starting points $\boldsymbol{X}_0$, and the corresponding sample points $\boldsymbol{X}_{60}$, after 60 steps of local optimization using Adam, based on the function c_1 in part (c) (see equation (73)). Additionally, one can see the autoencoded version of these points, with $AE_i(\boldsymbol{X}_{60}) = D_i(E_i(\boldsymbol{X}_{60}))$, with the thin line of the same color corresponding to the decoded latent space domain X_{Z_i}.
Meanwhile, in part (b), one can see the values of the cost function $c_1(D_i(z))$ over the latent space domain ($z \in Z = [0, 1]$). The optimum can here be found at the position $\boldsymbol{x}_{\min} = \underset{\boldsymbol{x}\in X}{\operatorname{argmin}}\ c_1(\boldsymbol{x}) = (0.75, 0)$, with $c_1(\boldsymbol{x}_{\min}) = 0$. See Appendix H for details on the cost function and training of the autoencoder.



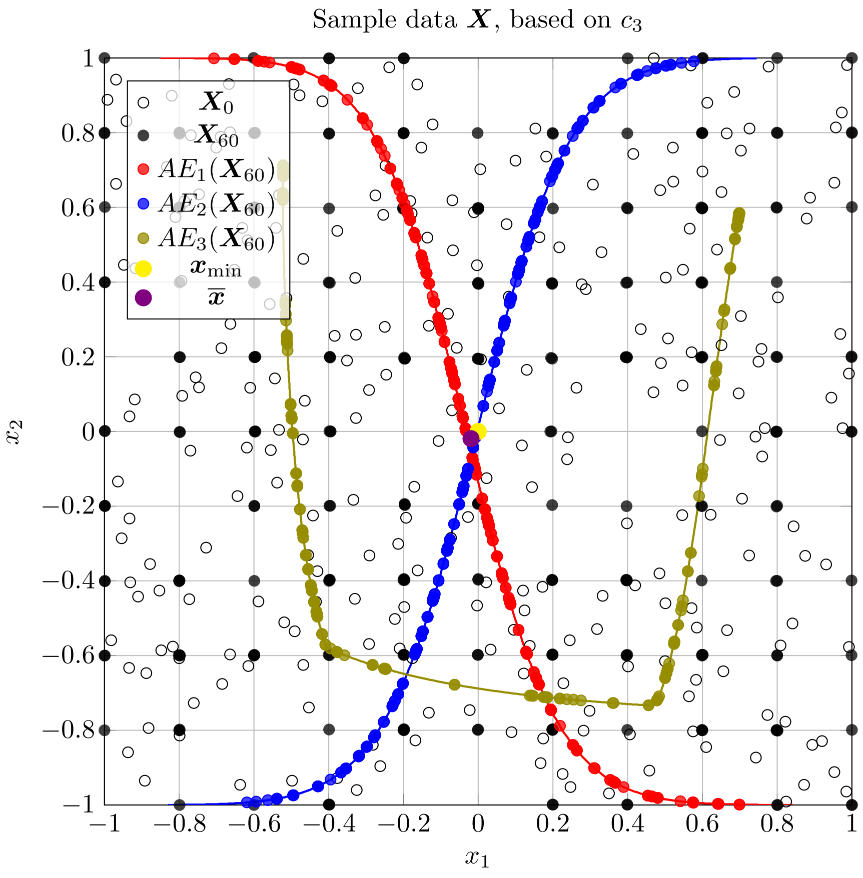
(a)



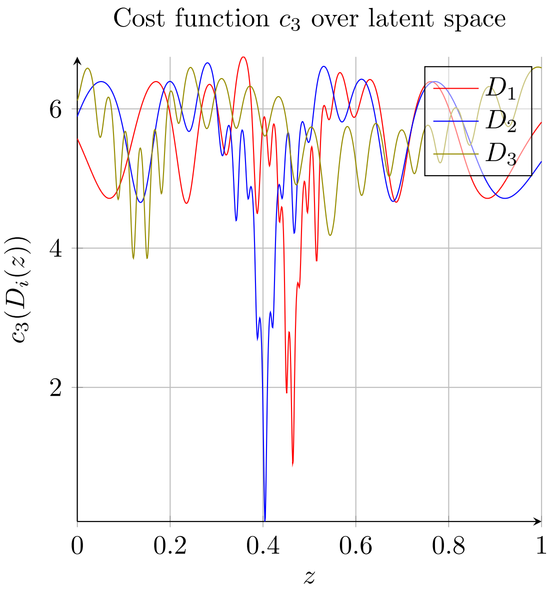


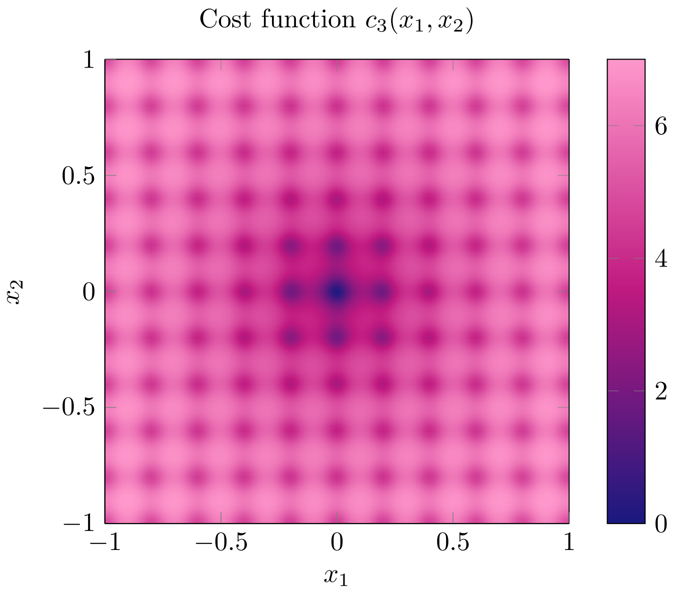
(b) (c)

Figure 16: The analogue to Figure 15, only for the cost function c_2 (see equation (73)). The optimum can here be found at the position $\boldsymbol{x}_{\min} = \underset{\boldsymbol{x}\in X}{\operatorname{argmin}}\, c_2(\boldsymbol{x}) = (-0.5, 0)$, with $c_2(\boldsymbol{x}_{\min}) = 0$. See Appendix H for details on the cost function and training of the autoencoder.



(a)





(b) (c)

Figure 17: The analogue to Figure 15, only for the cost function c_3 (see equation (73)). The optimum can here be found at the position $\boldsymbol{x}_{\min} = \underset{\boldsymbol{x}\in X}{\operatorname{argmin}}\ c_3(\boldsymbol{x}) = (0, 0)$, with $c_3(\boldsymbol{x}_{\min}) = 0$. See Appendix H for details on the cost function and training of the autoencoder.

following weights used in equation (45) to concentrate the autoencoder on the design boundaries:

$$[w_i]_{k,l} = \begin{cases} 1 & \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 1 \\ 0.5 & 1 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 2 \\ 0.25 & 2 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 5 \\ 0.1 & 5 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 10 \\ 0.025 & 10 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 20 \\ 0.01 & 20 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \end{cases} \tag{79}$$

Furthermore, 50 epochs of pretraining (see Appendix A.10) are used for the encoder network E and decoder network D, where the first 45 epochs use 10 batches each, and the last 5 epochs only consist of one batch each. During that, each part of the network will be split into $n_{\text{pre}} = 2$ parts, with the outer part consisting of the convolutional layers of the network, and the inner part being the fully connected layers (see Figure 40a).
After that, 225 epochs with 10 batches each and further 25 epochs consisting only of one batch are used to train the whole network (see Appendix A.9).
The use of these settings was used due to it being superior in its resulting design compared to other network setups used for the same problem. It has to be noted that, to the best of the author's knowledge, the combination of using a discriminator as well as a surrogate model is novel, the same being true for the use of the signed distance filed as a method of encoding the designs.

I.3 Optimization over latent space

When optimizing over latent space, the following cost function was optimized, in this case with $\mu = 25$:

$$c_\mu(\boldsymbol{z}) = \left(C \circ SIMP^{10}_{0.25,\{\}} \circ SIMP^{\mu}_{1,\{\}} \circ SDF^{-1} \circ D\right)(\boldsymbol{z}) \tag{80}$$

Here, the second part of local optimization using SIMP (this time with an smaller ϵ) is done to avoid the issues with the influence of gray values on the quality of a proposed design (see Appendix K.5). This is also the main motivation behind the inclusion of this part in the generation of the training set and the post-processing.
In this part, two different methods are applied to optimize the cost function c_{25}.

- Differential evolution $DE_{300,500,0.6,0.9}$ (see Appendix A.3)
- Bayesian optimization with the parameters $M_0 = 1000$ and $i_{BO} = 300$ and with $T = 1$ (see Appendix A.4). While Differential evolution was used to optimize the model parameters in every iteration of the optimization process, the method used was not always the same. Namely, one used $DE_{100,G_i,0.6,0.9}$ with

  $$G_i = \begin{cases} 50 & i \equiv 0 \mod 100 \\ 10 & \text{else} \end{cases} \tag{81}$$

  To emphasize exploration of the latent space in the beginning, and exploitation in the end, the following values for ξ_i in the acquisition function EI_i are selected (see Appendix A.4):

  $$\xi_i = \begin{cases} 10 & i < 200 \\ 0 & i \geq 200 \end{cases} \tag{82}$$

  Additionally, the acquisition function EI_i was maximized by minimizing $-EI_i$ using $DE_{100,500,0.6,0.9}$.

I.4 Post-processing

From the optimum $\boldsymbol{z}^*$, one can get the final design $\boldsymbol{x}^*$:

$$\boldsymbol{x}^* = \left(SIMP^{10}_{0.25,\{\}} \circ SIMP^{300}_{1,\{\}} \circ SDF^{-1} \circ D\right)(\boldsymbol{z}^*) \tag{83}$$

That means that the post processsing method from equation (6) is used.

I.5 Variations on the design

while in the final results, a adversarial autoencoder with a surrogate network was used to encode the signed distance field, with $m = 100$ and $\mu = 25$, other possibilities were tested to see which way would let to the best results, with the variations visible in Table 6: It has to be noted that for the case of using

Table 6: The different variation tested for solving the optimization problem.

i	1	2	3	4	5	6
$\delta_i = 0$	$m = 25$	no pretraining used	no D_{Dis} used	no S used	no SDF used	$\mu = 1$
$\delta_i = 1$	$m = 100$	pretraining used	D_{Dis} used	S used	SDF used	$\mu = 25$

not the signed distance field, but the density to encode the designs, the following weights where used in equation (45):

$$[w_i]_{k,l} = 2\left(1 - |\,[x_i]_{k,l} - 0.5|\right) \tag{84}$$

Each of these methods, which are all based on the same training data, are then done four times, to take into account the randomness involved in the initialization of the neural network parameters (see Appendix A.9) and the optimization over latent space (here, only differential evolution was used due to its greater speed (see Appendix K). The method is then deemed to be successful if it can reliably produce final results whose compliance C is lower then the lowest compliance from the training samples. From the 64 different test cases, this is the case in 14 cases. In Table 7 it can been how these successful cases are distributed between the different variations: These results also overlap (except in the case of δ_4, the use of

Table 7: The ratio of successful variations belonging to each instance of the varied variables, given in percent.

i	1	2	3	4	5	6
$\delta_i = 0$	28.6	21.4	35.7	50.0	35.7	0
$\delta_i = 1$	71.4	78.6	64.3	50.0	64.3	100

the surrogate model S) with the final set-up used ($\delta_i = 1\ \forall i$), which is able to produce the design with the lowest compliance (see Figure 4d). This same network design also shows the lowest average compliance of the results over all four runs, making the case for choosing it as the design used in the main text even stronger.
The fact that 30 of the test cases where never able to outperform the training set shows that even if the underlying assumption about the cost function seems to be true, one still needs to be careful when constructing the network used to approximate the space X'.

J On the evaluation time for the optimization over latent space

When trying to approximate the running time for optimization over latent space, there are three variables to consider. The first one is the cost function evaluation time T_c. The second one is the time needed for modeling in Bayesian optimization T_M (if only run on one processor core). and the third and last is the number of available processors N_p. For the used implementations of both the methods (see Appendix **??**), one can then calculate T_{DE} and T_{BO} (overhead of parallelization is neglected here):

$$\begin{aligned} T_{DE} &= \frac{\gamma G}{N_p} T_c = \frac{150000}{N_p} T_c \\ T_{BO} &= \left(\frac{M_0}{N_p} + i_{BO}\right) T_c + \frac{T_M}{N_p} = \left(\frac{1000}{N_p} + 300\right) T_c + \frac{5000000s}{N_p} \end{aligned} \tag{85}$$

It has to be noted that these equations are only valid if:

$$\frac{\gamma}{N_p}, \frac{M_0}{N_p} \in \mathbb{N}^+ \tag{86}$$

When considering the viability of using Bayesian optimization, two factors are important. The first one is $T_{c,0}$, the point after which Bayesian optimization becomes more efficient. This can only happen when the condition $\gamma G - M_0 - N_p i_{BO} > 0$ is fulfilled:

$$\begin{aligned} &T_c \geq T_{c,0} \iff T_{DE} \geq T_{BO} \\ &\Rightarrow T_{c,0} = \frac{T_M}{\gamma G - M_0 - N_p i_{BO}} = \frac{5000000s}{149000 - 300N_p} \end{aligned} \tag{87}$$

If $T_{c,0}$ indeed exists, then the second import information is the factor k_T:

$$k_T = \left.\frac{T_{DE}}{T_{BO}}\right|_{T_c \to \infty} = \frac{\gamma G}{M_0 + N_p i_{BO}} = \frac{150000}{1000 + 300N_p} \tag{88}$$

For $N_p = 100$, as used in this work, one will get $T_{c,0} \approx 42.0s$ and $k_T \approx 4.84$. If no parallelization is possible, then one will get $T_{c,0} \approx 33.6s$ and $k_T \approx 115.4$.
If there exists an efficient parallelization of the cost function itself, with a negligible overhead, then the problem becomes similar to $N_p = 1$.

K Compliance minimization 2

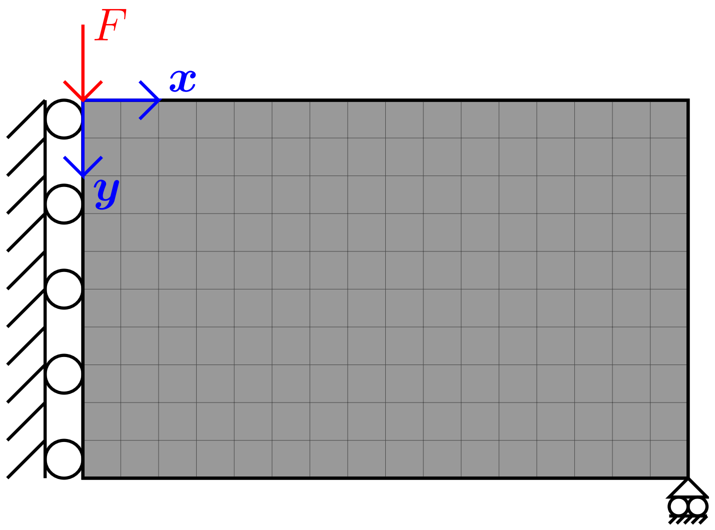


Figure 18: Problem that has to be solved, in this case with $n_y \times n_x = 10 \times 16 \Rightarrow n = 160$, where n_y and n_x are the number of quadratic FEM elements in the respective directions.

The proposed method will be tested for the (relatively simple) problem of minimizing the compliance C (see Figure 18):

$$\begin{gathered} \min_{\boldsymbol{x} \in [0,1]^{n_y \times n_x}} C(\boldsymbol{x}) = \langle \boldsymbol{u}(\boldsymbol{x}), \boldsymbol{F} \rangle \\ \boldsymbol{K}(\boldsymbol{x})\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{F} \\ \overline{\boldsymbol{x}} \overset{!}{=} \rho_0 = 0.4 \end{gathered} \tag{89}$$

Here, $\boldsymbol{K} \in \mathbb{R}^{2n_y n_x \times 2n_y n_x}$ is the stiffness matrix resulting from the mesh, and $\boldsymbol{F}$ is the force vector, while $\boldsymbol{u}$ is the displacement field. $\boldsymbol{F}$ is here a vector, where the only non-zero element corresponds to the downward force F in the upper left corner (see red arrow in Figure 18). The compliance $C(\boldsymbol{x})$ corresponds to the cost function $c(\boldsymbol{x})$.

K.1 Varying network types

In a first test, for three of the four steps of the proposed method (see section 1.2), a total of eight hyperparameters are varied.

1. Generation of training data (see Appendix L.1 for detailed implementation)
    - The number of elements n_y and n_x used in the problem, with $n_y \times n_x \in \{30 \times 60, 45 \times 90, 60 \times 120\}$.
    - The number of local optimization steps λ being used to create the training samples, with $\lambda \in \{0, 30, 300\}$.
2. Training of the Autoencoder (see Appendix L.2 for detailed implementation)
    - The number of latent space dimensions m of the autoencoder, with $m \in \{25, 50\}$.
    - The use of a density field or a signed distance field SDF to represent a design during training (see Appendix A.8).
    - The use of a discriminator network D_{Dis} to enforce a certain distribution in latent space. Using this network would result in an adversarial autoencoder [61] (see Appendix A.7).
    - The use of a surrogate model network S, which maps the latent space representation $E(\boldsymbol{x})$ of an sample $\boldsymbol{x}$ to its cost function value $c(\boldsymbol{x})$ (see Appendix A.7).
    - The use of pretraining in the autoencoder (see Appendix A.10).
3. Optimization over latent space (see Appendix L.3 for detailed implementation)
    - The cost function c_μ which is used to optimize in latent space, with $\mu \in \{1, 10\}$. Here, C_0 is not used to avoid problems of violating the volume constraint ($\overline{\rho} = 0.4$). In a third run, a few random points in latent space are generated and then optimized using SIMP. This is mainly used to see if global optimization in latent space is advantageous.
4. Post-processing (see Appendix L.4 for detailed implementation) will have no variations.

Finally, as there is a lot of randomness involved in the whole process (for example the initialization of the network parameters, or in the optimization over latent space). To limit the influence of the randomness, for each training set and variance of autoencoder, two training runs will be performed. Both of the decoders produced are than used for optimization.
This will allow to estimate if the different performance of different setups tested in this work are caused by the differences in these setups, or if they are only due to the randomness mentioned above.
The networks used for these simulations are shown in the Figures 41, 42, 43, and 44

Results

The first possible point to be considered are the mean squared reconstruction errors E_{train} produced during the training of the autoencoders, and especially the comparison to the reconstruction error E_{val} from the validation samples not considered during training. As it can be seen in Figure 19, in no case the validation error is larger than $1.01 E_{\text{train}}$. Consequently, overfitting does not seem to be a problem, which also indicates that network training might be performed with fewer parameters or networks with more trainable parameters could be trained on the same training set.
Additionally, it can also be observed that an increase in λ generally leads to a decrease in the reconstruction error, although this is not the case for every network trained.
But more important in this work is the ability of the decoder to produce good, stiff designs which can be found reliably during optimization. Here, the results can be seen in the Figures 20, 21, 22, 23, and 24, where a number of trends for each of the variations pursued can be observed.

- Firstly, it seems clear that a increase in problem size seems to be negatively impacting the ability of the decoders to produce superior designs. For example, the best design in $n_y \times n_x = 30 \times 60$ has a compliance of $C \approx 0.953 C_h \approx 0.961 C_r$ (C_h is the compliance of the design achieved after 1000 steps of SIMP, starting from a homogeneous design, while C_r is the compliance of the best design in $\boldsymbol{X}_\lambda$). Meanwhile, the best design in $n_y \times n_x = 60 \times 120$ has a compliance of $C \approx 0.991 C_h \approx 0.994 C_r$. Additionally, while the majority of all designs for $n_y \times n_x = 30 \times 60$ and $\lambda > 0$ can outperform C_r, and C_h is worse in every case, this is rarely achieved in the case of $n_y \times n_x = 60 \times 120$.

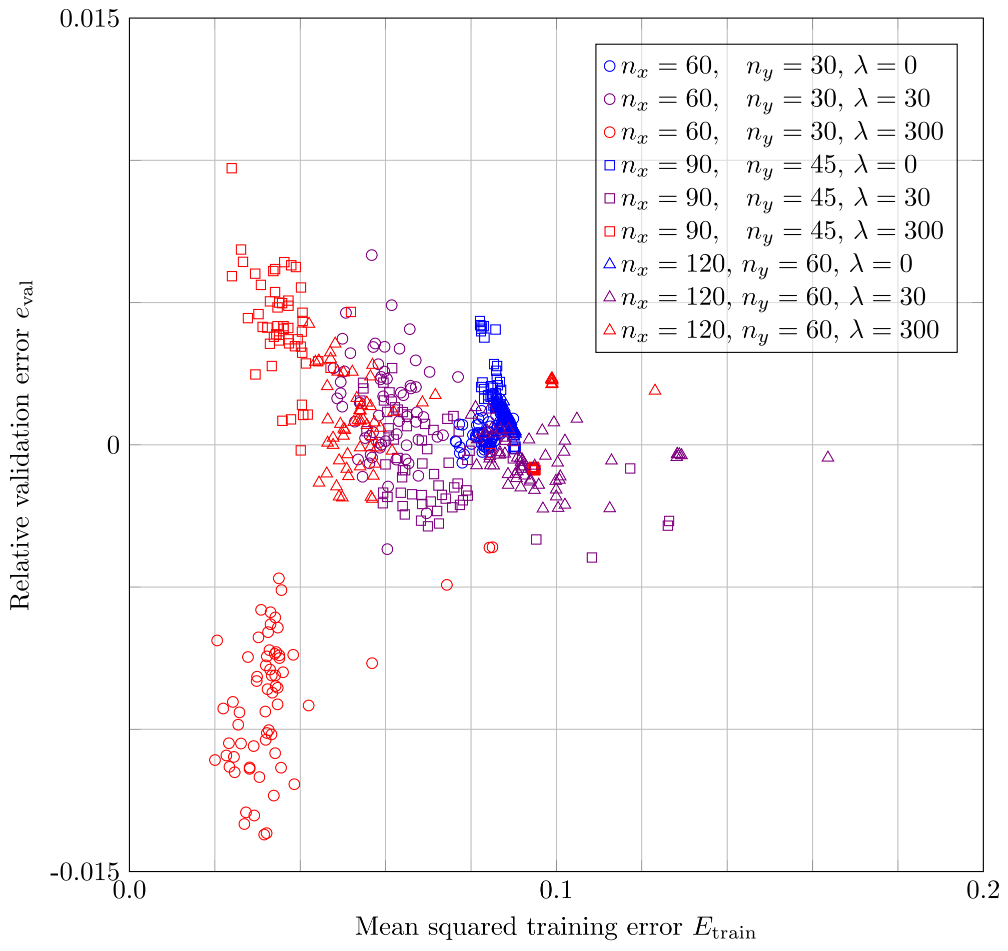


Figure 19: Relative validation error $e_{\text{val}} = \frac{E_{\text{val}}}{E_{\text{train}}}$ over the training error. The small values indicate that overfitting does not seem to be a problem.

- When looking at the influence of λ, it can be seen that the best designs are found for $\lambda = 300$, in which case the average minimum compliance also is the lowest (see Table 8).

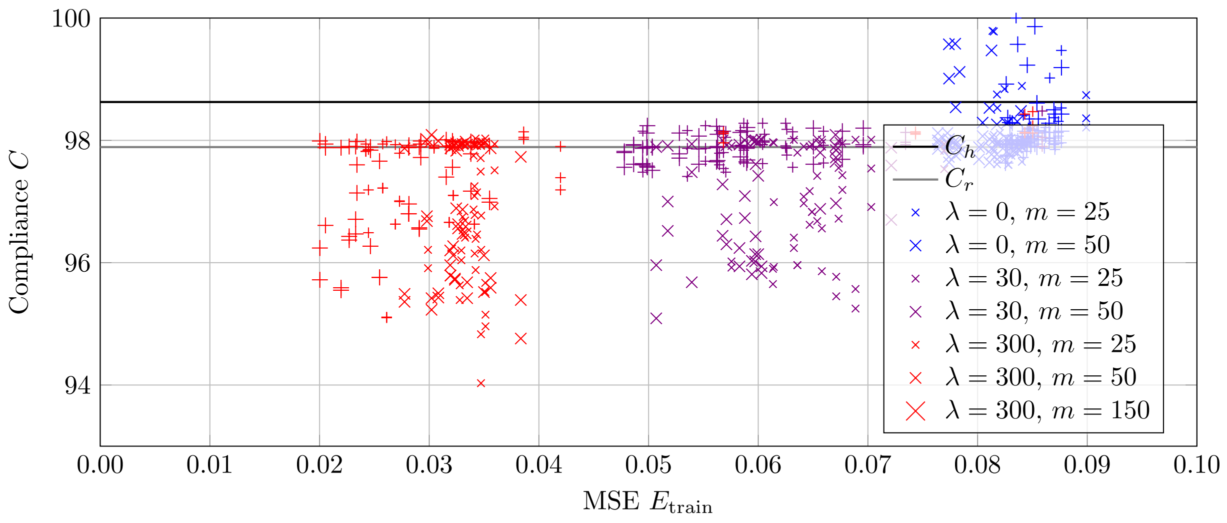
Table 8: Minimum compliance $C_{\min}$ and mean compliance $\overline{C}$ in dependence of $n_y \times n_x$ and λ, in the form $C_{\min}/\overline{C}$.

$n_y \times n_x$	$\lambda = 0$	$\lambda = 30$	$\lambda = 300$
30×60	97.69/98.54	95.09/97.48	94.03/97.07
45×90	95.07/96.10	94.49/95.61	94.33/95.44
60×120	94.19/94.93	94.19/94.91	93.67/94.47

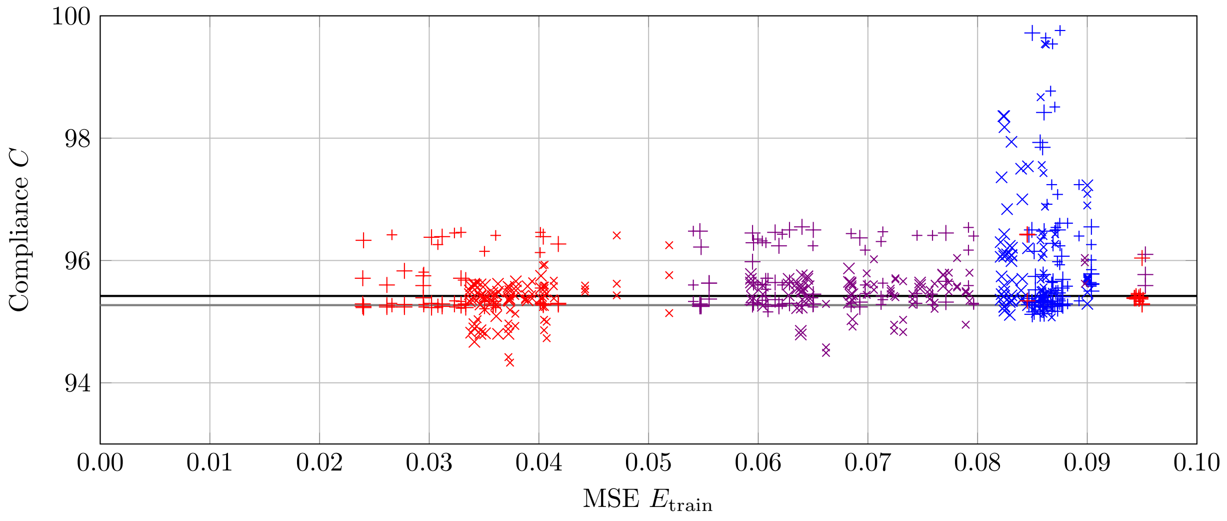
As the use of a high λ seems to be beneficial, the cases of $\lambda \in \{0, 30\}$ will be neglected in the further analysis of the results.

- When looking at the number of latent space variables m, it can be seen that a higher m generally allows for a lower reconstruction error E_{train}, but the influence on the compliance found seems to be minimal, with no clear trend. For every problem size, the best design is always found on a decoder with $m = 25$, but there is no trend to generally infer that a lower m leads to better designs.

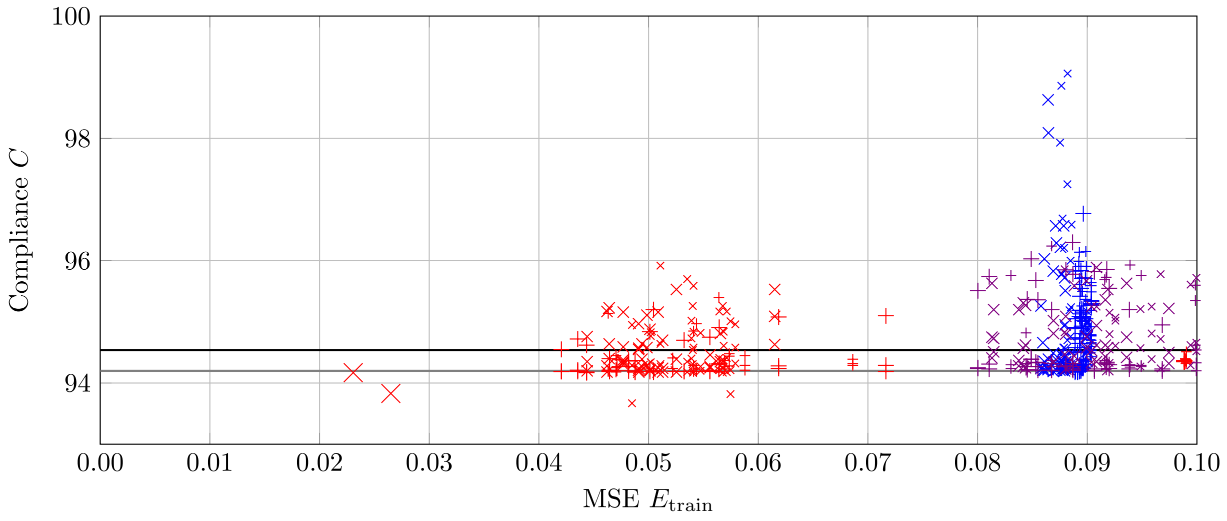
- When looking at the difference between using a density field and a signed distance field in Figure 20, the use of a signed distance field seems to be beneficial, as all of the best designs use this kind of formulation. For example, in the case of $n_y \times n_x = 30 \times 60$ and $\lambda = 300$, in nearly four out of five cases, *SDF* is the better choice, including the best design. Additionally, in the other problem



(a) $n_y \times n_x = 30 \times 60$

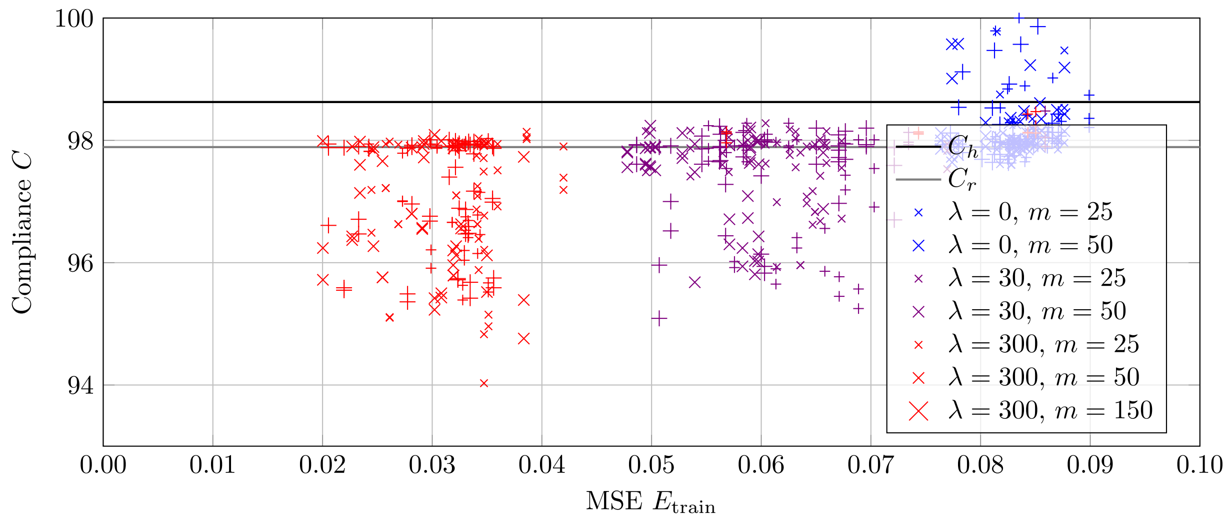


(b) $n_y \times n_x = 45 \times 90$

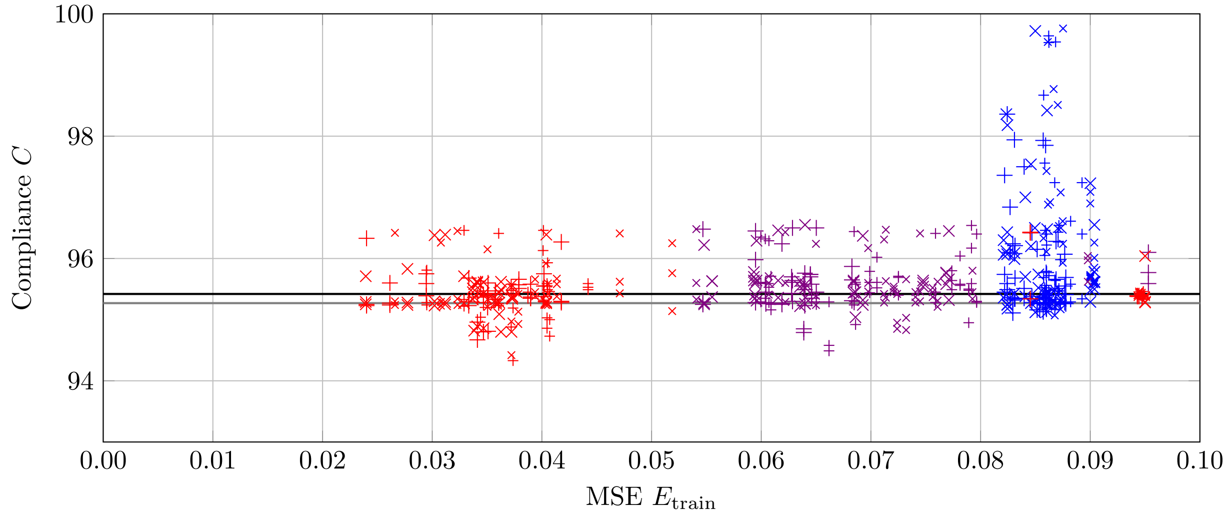


(c) $n_y \times n_x = 60 \times 120$

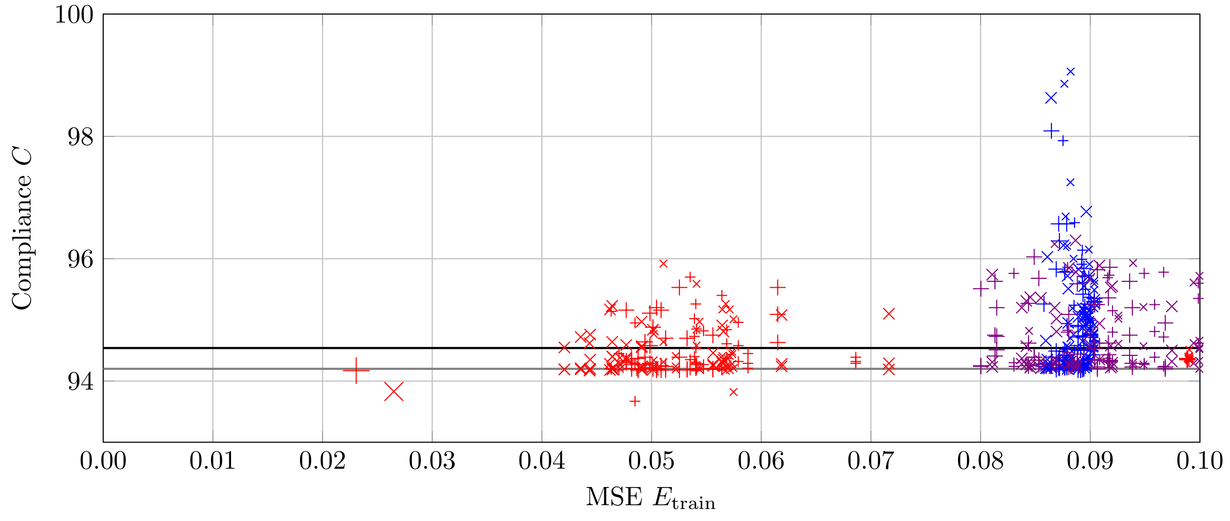
Figure 20: Mean squared error reconstruction error E_{train} and compliance C after post-processing for all the different designs used in section K.1. Autoencoders trained using a signed distance field are symbolized by crosses, and ones trained on a density field by pluses.



(a) $n_y \times n_x = 30 \times 60$

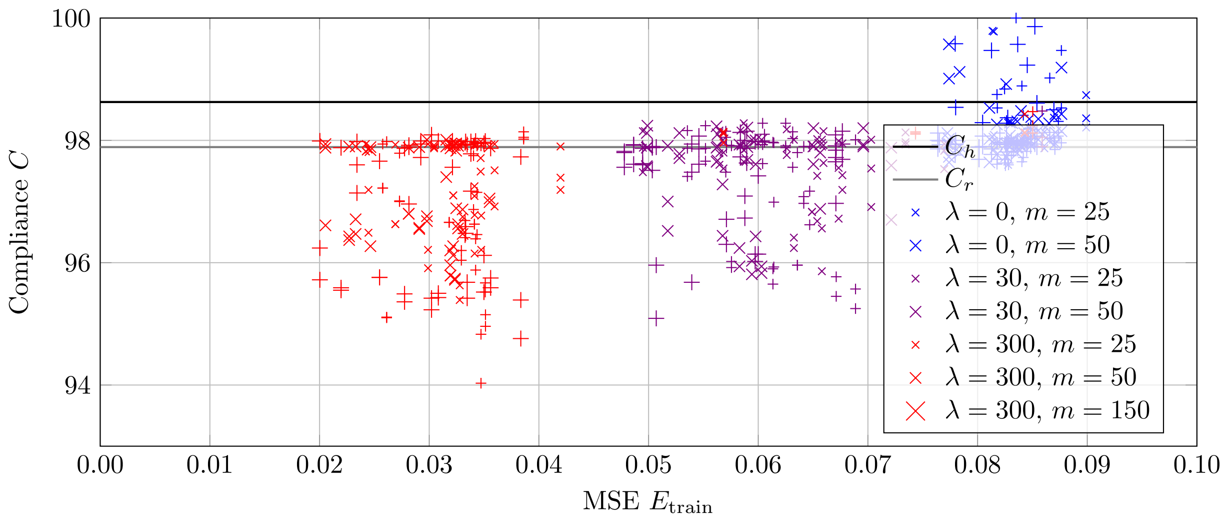


(b) $n_y \times n_x = 45 \times 90$

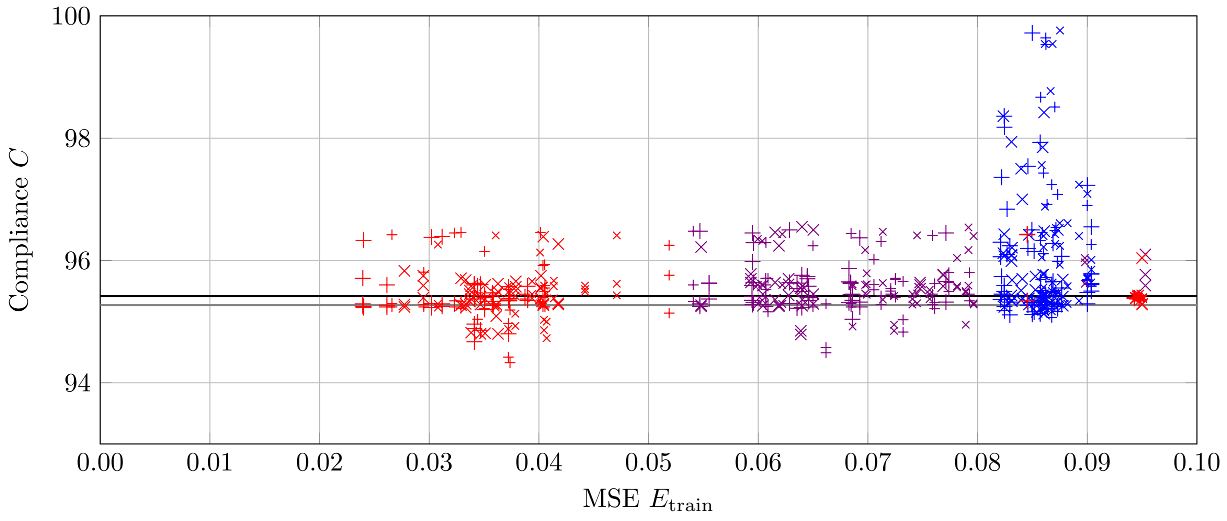


(c) $n_y \times n_x = 60 \times 120$

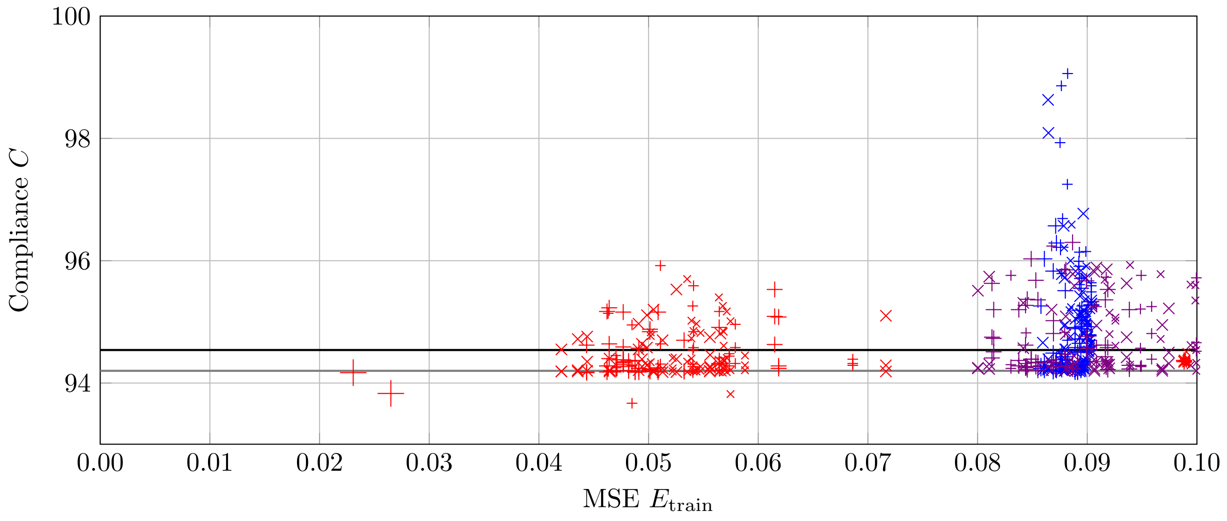
Figure 21: Mean squared error reconstruction error E_{train} and compliance C after post-processing for all the different designs used in section K.1. Autoencoders trained using the surrogate network D_{Dis} are symbolized by crosses, and ones trained without by pluses.



(a) $n_y \times n_x = 30 \times 60$

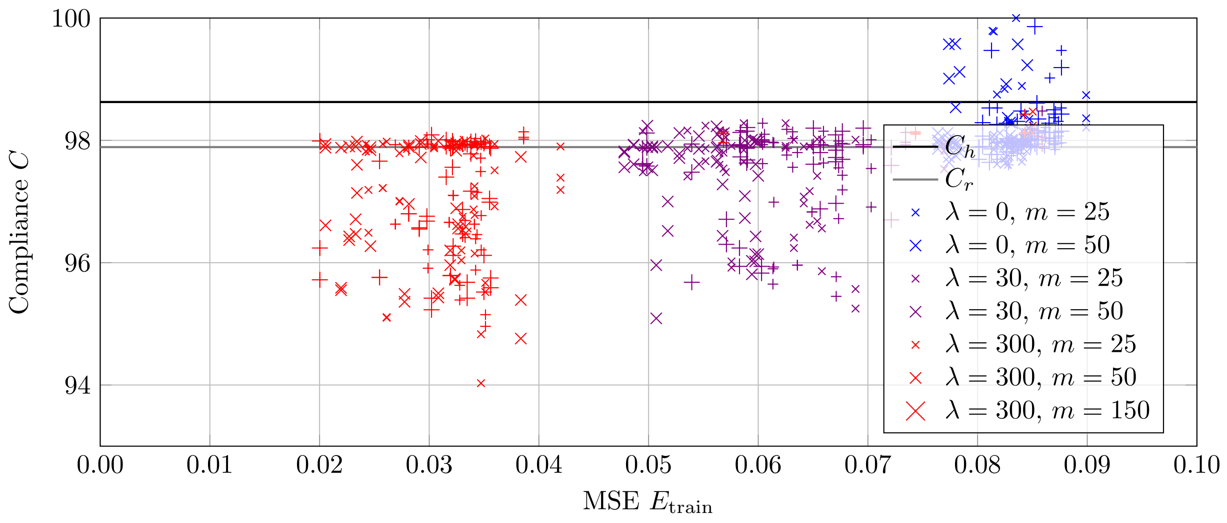


(b) $n_y \times n_x = 45 \times 90$

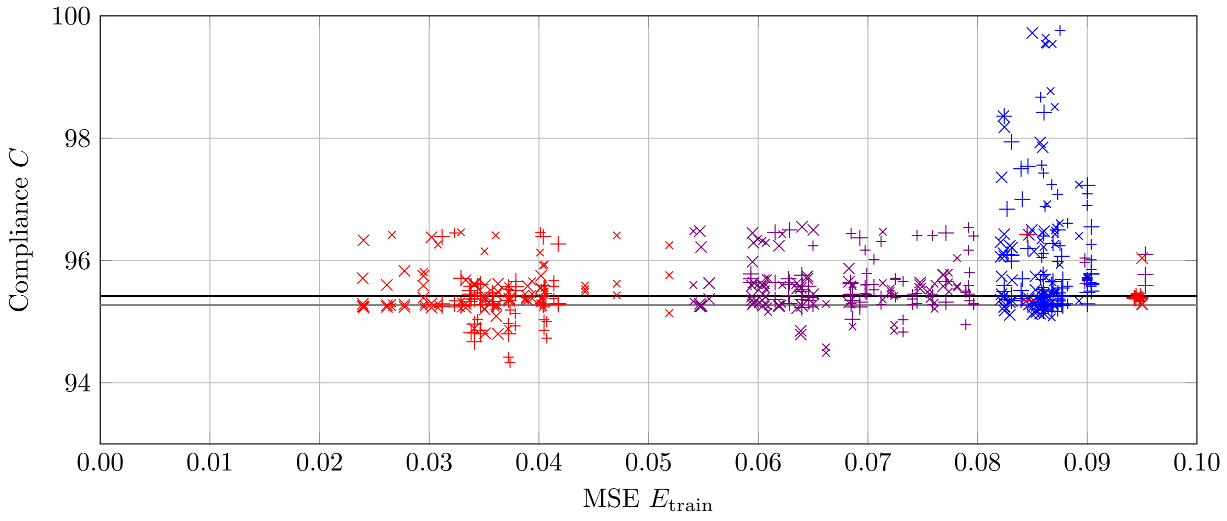


(c) $n_y \times n_x = 60 \times 120$

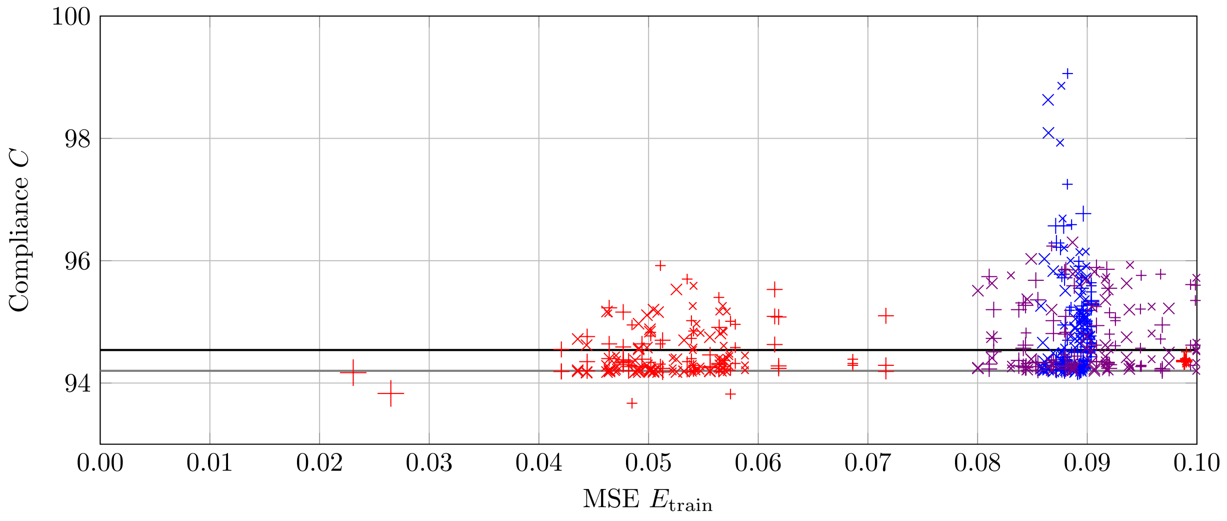
Figure 22: Mean squared error reconstruction error E_{train} and compliance C after post-processing for all the different designs used in section K.1. Autoencoders trained using the surrogate network S are symbolized by crosses, and ones trained without by pluses.



(a) $n_y \times n_x = 30 \times 60$

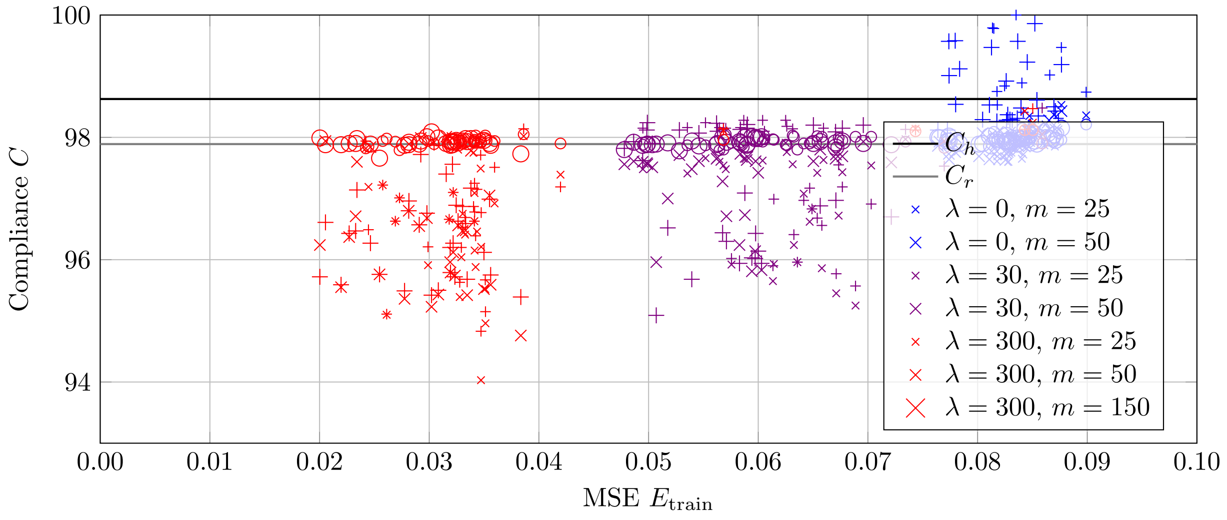


(b) $n_y \times n_x = 45 \times 90$

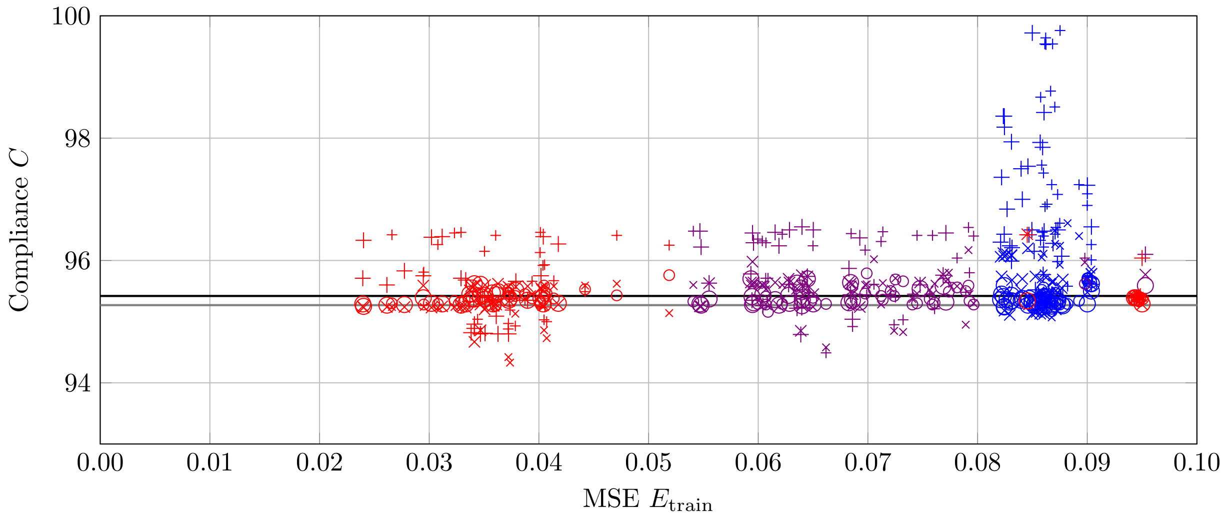


(c) $n_y \times n_x = 60 \times 120$

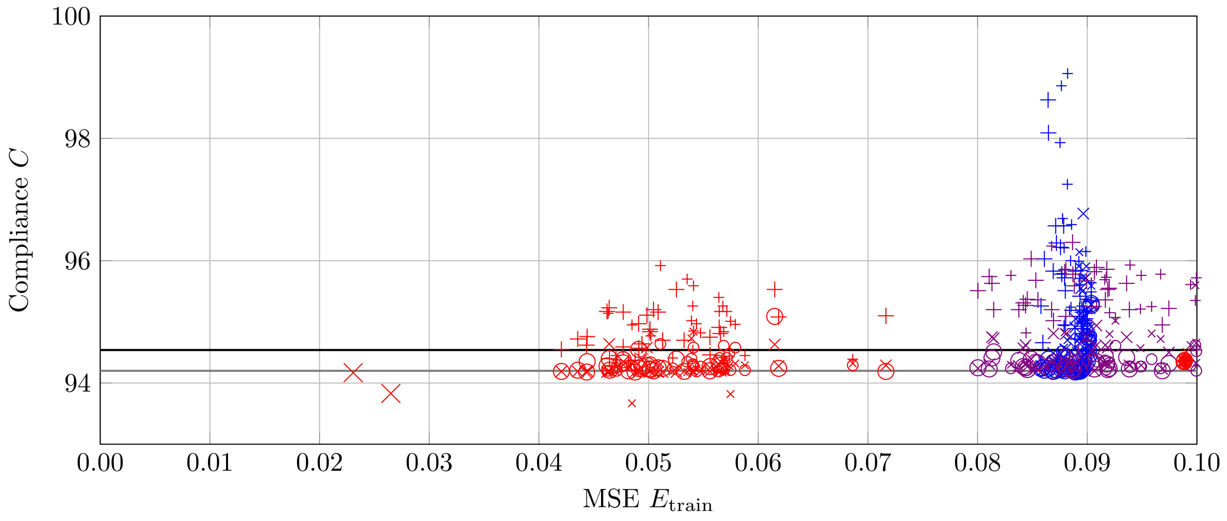
Figure 23: Mean squared error reconstruction error E_{train} and compliance C after post-processing for all the different designs used in section K.1. Autoencoders trained with pretraining are symbolized by crosses, and ones trained without by pluses.



(a) $n_y \times n_x = 30 \times 60$



(b) $n_y \times n_x = 45 \times 90$



(c) $n_y \times n_x = 60 \times 120$

Figure 24: Mean squared error reconstruction error E_{train} and compliance C after post-processing for all the different designs used in section K.1. Optimization done with $\mu = 1$ are symbolized by pluses and ones with $\mu = 10$ with crosses. Circles represent the cases using random latent space samples without differential evolution as starting points fro post-processing.

sizes, the use of SDF seems to be the only method to outperform C_r. Therefore, it seems overall advantageous to use the signed distance field during network training and latent space optimization.

- When looking at the possible use of the discriminator network D_{Dis} in Figure 21, the influence seems to be much smaller. In the case of $n_y \times n_x = 30 \times 60$ the use of the discriminator seems to be advantageous, with using it being better in $\approx 65\%$ of cases, including most of the best designs. But this trend cannot be found for the cases with larger problem sizes. Therefore, using a discriminator or not seems to have little influence on the final design found. As the training of the discriminator takes time, it is therefore likely better if it is not used.

- Similarly, no clear trend can be seen in regard to the use of a surrogate network S in Figure 22, although especially in the case of $n_y \times n_x = 30 \times 60$, the use of a surrogate network seems to be a obstacle in regard to finding good designs. As the training of the surrogate network again takes extra time, with seemingly no resulting benefit, not using it likely is the better choice.

- In regard to pretraining, there is a split trend. As can be seen in Figure 23, not using pretraining seems to be the most promising approach in outperforming C_r for $n_y \times n_x = 45 \times 90$ and $n_y \times n_x = 60 \times 120$. But on the other hand, in the case of $n_y \times n_x = 30 \times 60$, using pretraining seems to be beneficial, with most of the best designs relying on it. But as this proposed method should primarily solve larger problems, it seems that overall the use of pretraining seems to bring no benefit, while again costing time and computational capacity. Therefore, not using pretraining is likely the better choice.

- Lastly, the influence of μ can be looked at in Figure 24. The first point that can be noted here is that the use of latent space optimization is indeed necessary, as the random sampling approach in very few cases outperforms C_r, and in no cases produces a design with a compliance smaller than $0.9975\, C_r$.
  On the other hand, using latent space optimization can lead to significantly better results. Overall, in around 90% of cases, $\mu = 10$ leads to better results than $\mu = 1$, and additionally, all of the best designs for each problem size have been found using $\mu = 10$. Therefore, the use of $\mu = 10$ seems to be the better approach.

Overall, it has to be mentioned that there is no specific network layout which can reliably outperform C_r. Consequently, if used, it is likely that multiple runs of the method are required to improve upon the C_r. Additionally, when compared to standard SIMP, the proposed method takes a far longer time. While C_h is created using 1000 steps of local optimization LO, the superior designs are found only after hundreds of thousands of steps of local optimization and cost function evaluations, not to mention the training time needed for the neural network.
On another note, for each case with a specific problem size and a certain λ, there does not seem to be any trend in regard to the influence of the reconstruction error E_{train} on the compliance of the design produced by the specific network.

K.2 Varying the network layer design

The influence of the network design (number of layers L, and the number of nodes in each layer, ...) likely influences the ability of the network to produce good designs and will be looked at next. To test that, for one case ($n_y \times n_x = 30 \times 60$, $\lambda = 300$, $m = 25$, use of pretraining, a discriminator network D_{Dis} and a signed distance field via SDF, but no surrogate model S (see Appendix L.2 for details)), different network types for the encoder E and decoder D are tested, with the variations focusing on the three-dimensional layers of the network.

1. The filter size $t_y^{(l)}$ and $t_x^{(l)}$ is varied in many layers (except in layer 1 and L, which are determined by previous layers), with $t^{(l)} = t_y^{(l)} = t_x^{(l)} \in \{2, 3, 4, 5, 6\}$.

2. The stride $s^{(l)} = s_y^{(l)} = s_x^{(l)} \in \{2, 3\}$ is varied.

3. The channel number $\kappa^{(l)}$ is changed. Here, the channel number is either increased incrementally from layer to layer, or every three-dimensional layer (except the two outermost most) will assume the number of channels of the innermost three-dimensional layer.

The exact 20 networks used can be seen in Figures 45, 46, 47, 48, and 49. For each network, an optimization is performed for $\mu \in \{1, 10\}$, with a following post-processing. Additionally, 500 random points are selected from the latent space as a start for local optimization of 500 steps of SIMP in design space for each decoder trained.
Like in the previous section, every network is trained twice, with each being used for separate optimization. This is done to reduce the influence of the random steps in the process on the results. Again, this is done to reduce the amount of randomness in the results.

Results

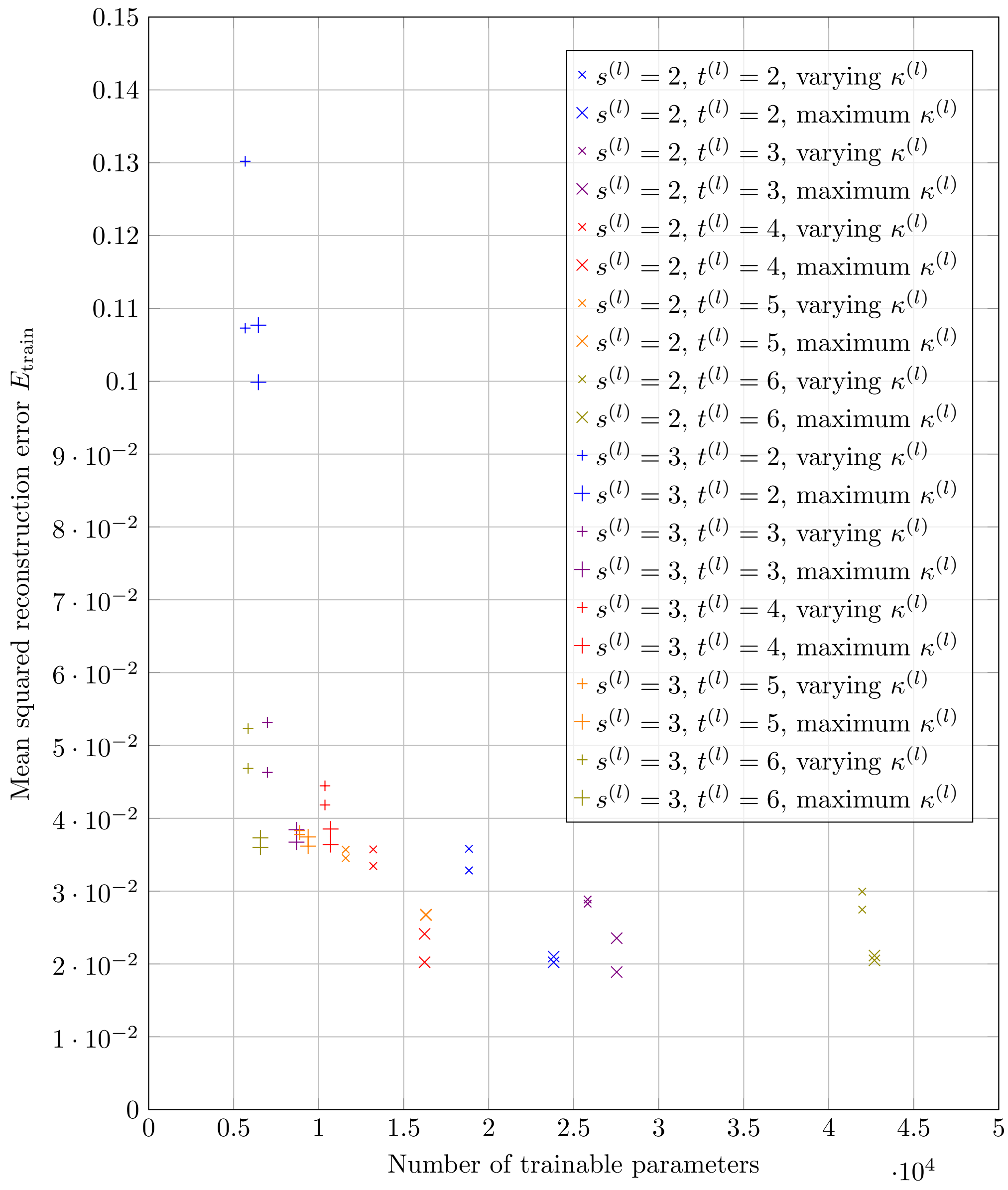


Figure 25: Reconstruction error for different network setups (see Figures 45, 46, 47, 48, and 49).

In the Figures 25, 26, 27, and 28, one can see a number of trends. The first to note would be the superiority of lower stride values, which in every instance produce networks with a lower reconstruction error and the ability to produce better designs before and after post processing. It might be proposed that a stride of $s^{(l)} = 1$ would be even better, but such networks are often too large to train in a feasible amount of time.
A second point to be noted is that using the maximum value of $\kappa^{(l)}$ in most three-dimensional layers,

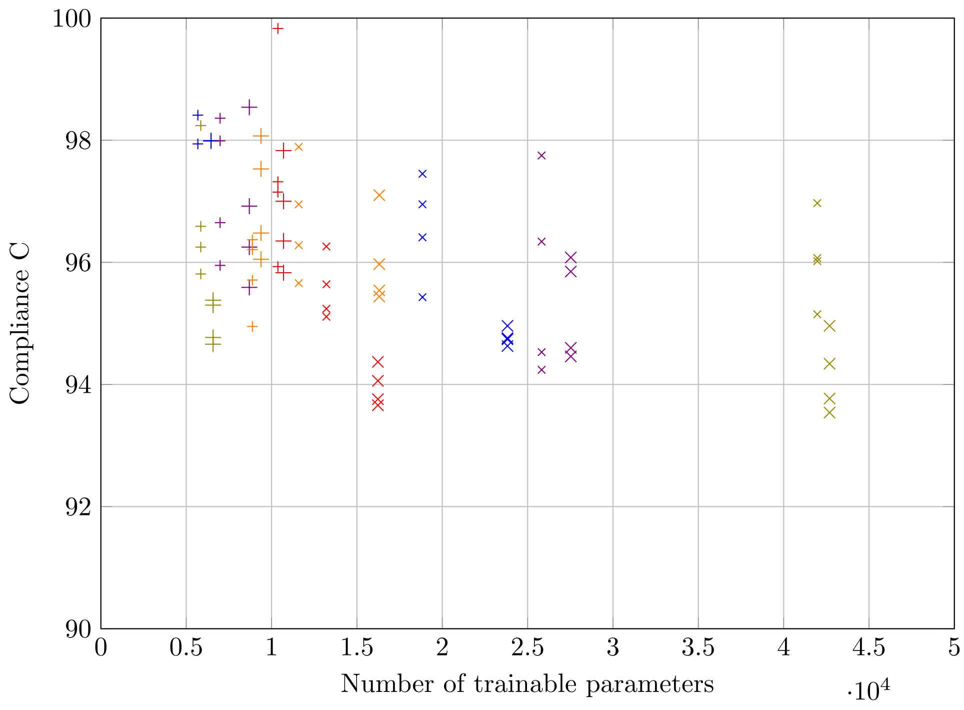


Figure 26: Best compliance before post-processing for different network setups (see Figure 25 for legend). Some values have compliance values above $C = 100$ and are therefore not visible in this figure.

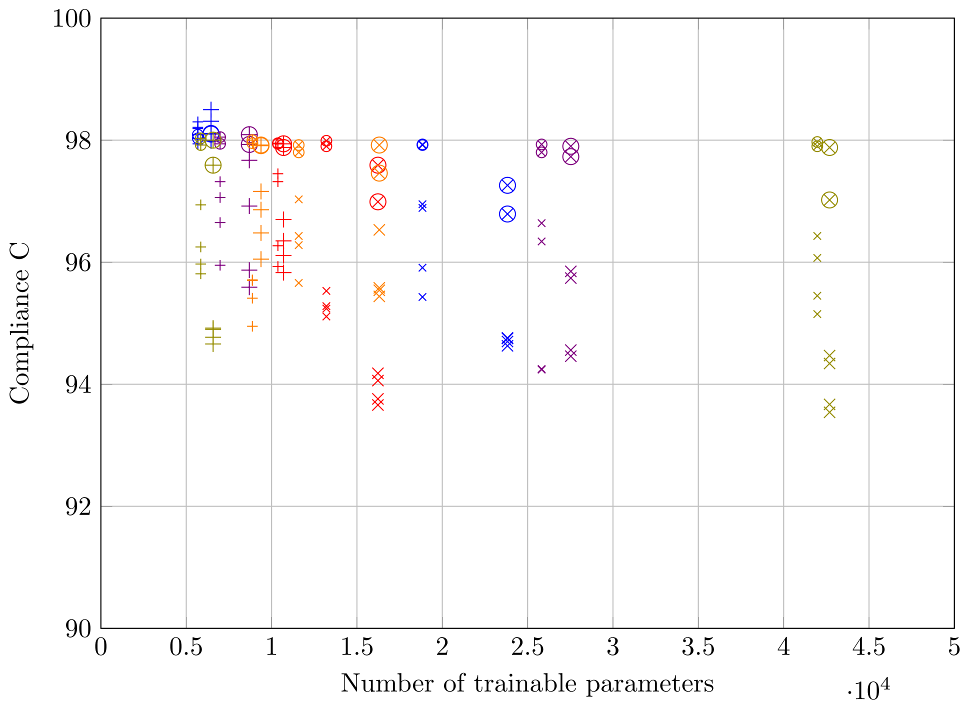


Figure 27: Best compliance after post-processing for different network setups (see Figure 25 for legend). The encirceld markers correspond to the designs gotten when not using optimization over latent space, but instead using post-processing on random latent space samples.

instead of increasing it towards the center of the network, will lead to a lower reconstruction error in every case and less compliant designs after optimization in most cases ($t^{(l)} = 3$ does not match here). When taking these two points out of consideration, it can be seen that changing the number of parameters

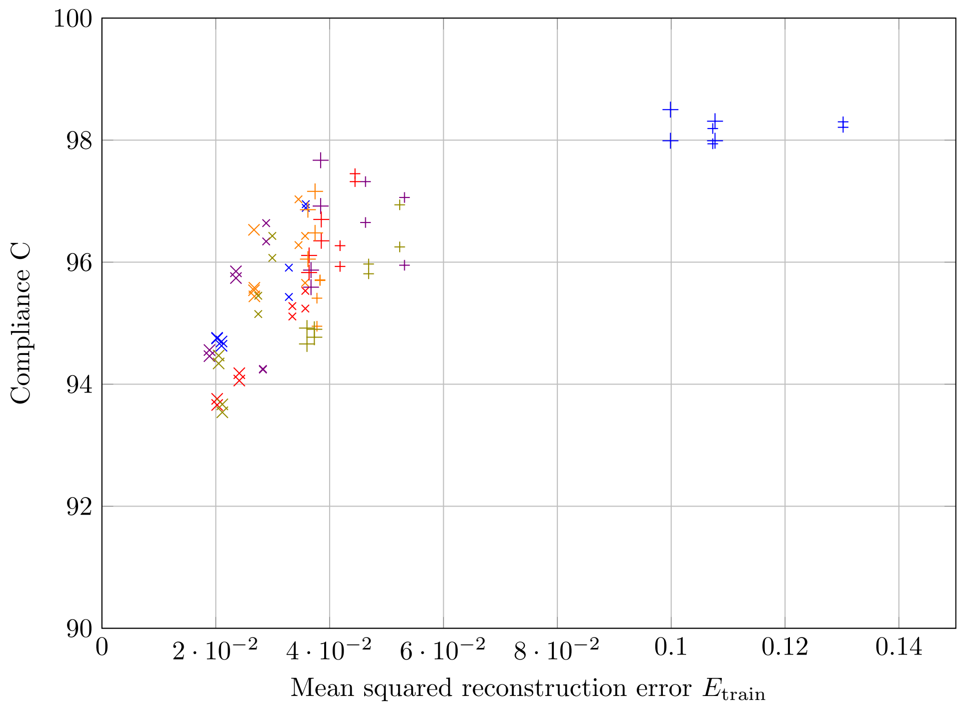


Figure 28: Compliance after post-processing over the reconstruction error of the network used for optimization (see Figure 25 for legend).

in another way (more parameters connecting the dense, inner layers of the network) will not drastically change the quality of the network, neither in regard to reconstruction error nor the compliance of the optimized designs.
Furthermore, it has also to be noted that the filter size seems to have a limited influence on the quality of the final design, although a filter size of $t^{(l)} = 4$ seems to be most efficient. It can also be seen, like in the previous part, that the use of optimization over latent space seems to be beneficial, as just post-processing random designs sampled in latent space does not lead to the same results.
Lastly, there seems to be a trend of networks with lower reconstruction error producing better designs (see Figures 28), which stands in contrast to the results from the previous simulations.
For all the results it can be noted that they are only valid for the specific circumstances of this test (mainly $n_y \times n_x = 30 \times 60$), so the same results might not appear in other cases.

K.3 Bayes optimization

When using an autoencoder to reduce the dimensionality of a topology optimization problem in the previous sections, differential evolution was used to optimize over latent space. As this process requires hundreds of thousands of cost function evaluation, using differential evolution might be too expensive if the cost function is hard to calculate. In such a case, one than can use Bayesian optimization (see Appendix A.4) to drastically reduce the number of required function evaluation.
To test the viability of such an approach, two decoders will be used, trained upon X_{300}, with pretraining and the use of a discriminator network D_{Dis}, but no surrogate model S, and the use of a signed distance field for $n_y \times n_x =\in \{30 \times 60, 60 \times 120\}$, with $m = 25$ (see Appendix L for details of producing this decoder). These decoders are used to map a latent space variable to a design in the design space during optimization of the cost function $\boldsymbol{c}_\mu$ from equation (96).
During testing, two variations will be looked upon:

1. The expense of the cost function $c_\mu(\boldsymbol{z})$ will be varied by choosing different $\mu \in \{1, 10, 100\}$. For each μ, differential evolution $DE_{100,500,0.6,0.9}$ will be used to optimize over latent space to provide a comparison to the results achieved by Bayesian optimization.

2. The frequency of full model updates during optimization will be changed, with full updates happening every $T \in \{5, 50\}$ iterations.

When using Bayesian optimization, the parameters $M_0 = 200$ and $i_{BO} = 100$ are used, with T being varied (see above). When using a full update ($i \equiv 0 \mod T$), $DE_{100,50,0.6,0.9}$ is used, while otherwise Adam with $\alpha = 0.005$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ is used for 10 steps.
To emphasize exploration of the latent space in the beginning, and exploitation in the end, the following values for ξ_i in the acquisition function EI_i are selected (see Appendix A.4):

$$\xi_i = \begin{cases} 5 & i < 50 \\ 0 & i \geq 50 \end{cases} \tag{90}$$

Additionally, the acquisition function EI_i was maximized by minimizing $-EI_i$ using $DE_{100,500,0.6,0.9}$.

Results

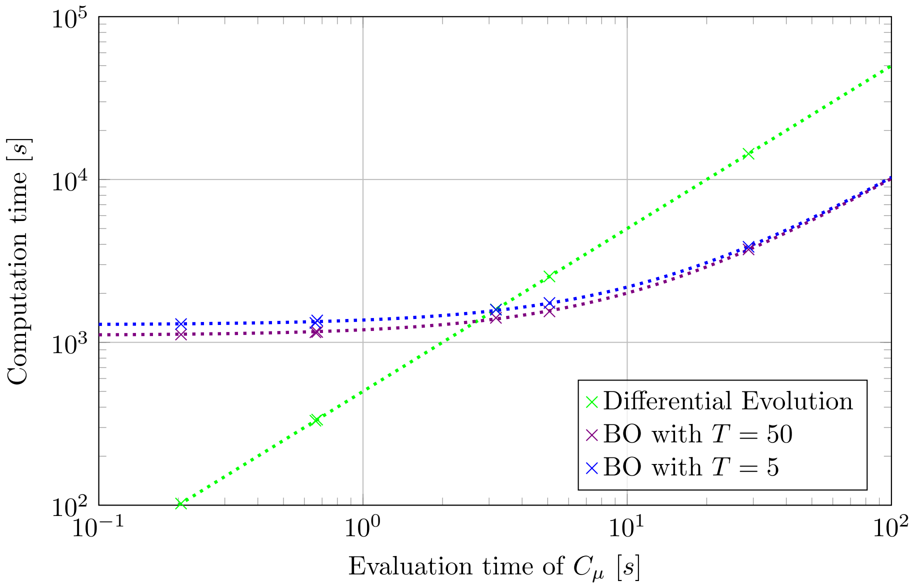


Figure 29: The time needed for different optimization algorithms. The dotted lines represent a linear regression.

The two aspects that have to be considered in regard to the usefulness of Bayesian optimization are the time needed to find an optimum and how good the found design is. It can be seen in Figure 29 that the time needed for the test cases optimized is significantly larger for Bayesian optimization than the standard approach (in this case differential evolution) for cheap cost functions. But for cost function with an evaluation time $\gtrsim 4s$, Bayesian optimization will be quicker, if the problems dimensionality is still $m = 25$.
For different m, this point will likely vary, as for larger m, a higher M_0 needs to be chosen and fitting the surrogate model will become more expensive. But on the other hand, increasing m also will likely necessitate more function evaluation, so predicting the exact point where BO becomes cheaper is difficult (see Appendix J for an example with $m = 100$).
It has also to be added that parallelization is used for the differential evolution during the test, running all $\gamma = 100$ function evaluations of one generation simultaneously. Therefore, if no parallelization would be feasible, due to a lack of computational power, the time needed for differential evolution would increase nearly by two orders of magnitude. A disadvantage of Bayesian optimization is that the method is not able to parallelize the evaluation of the cost functions as each evaluation depends iteratively on the previous one.

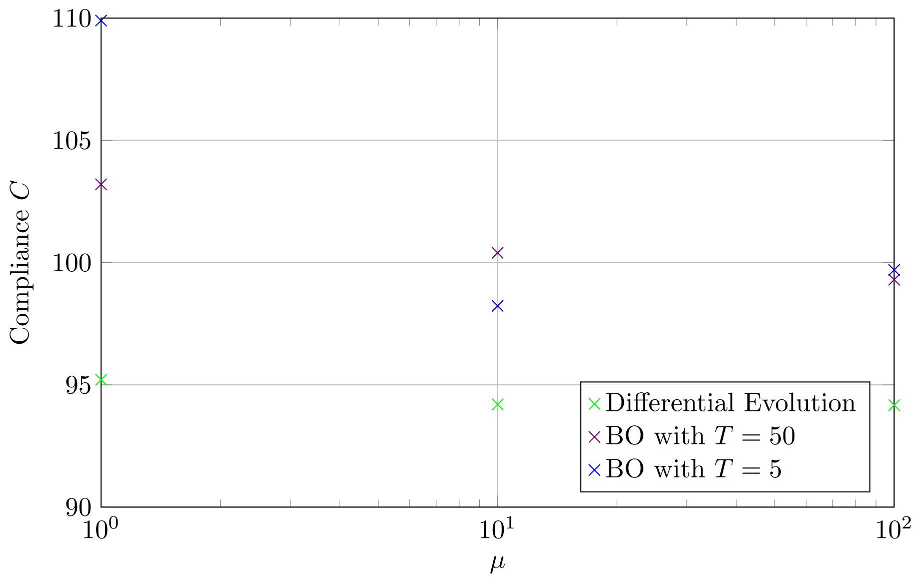


Figure 30: The minimum compliance found for different optimization algorithms (without post-processing), for $n_y \times n_x = 30 \times 60$.

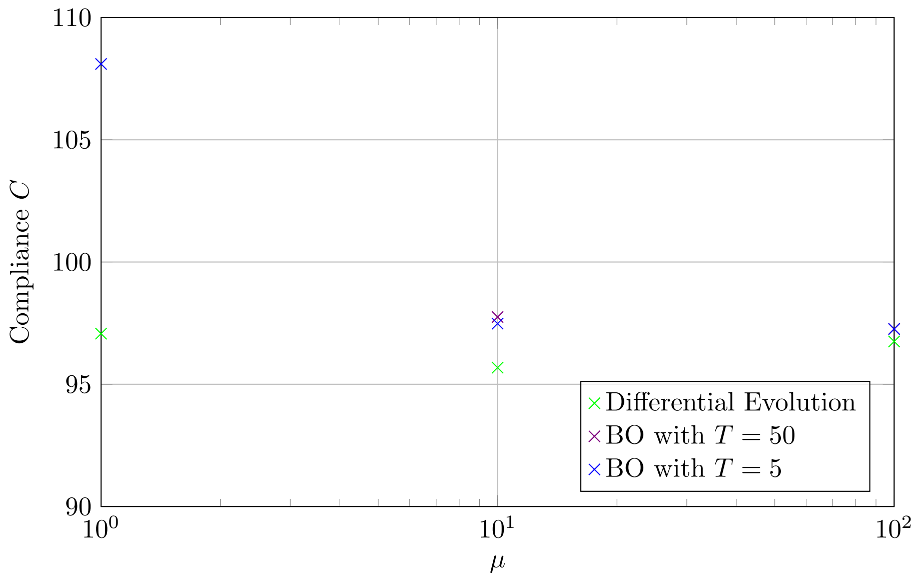


Figure 31: The minimum compliance found for different optimization algorithms (without post-processing), for $n_y \times n_x = 60 \times 120$.

But in regard to the quality of the optimized designs found, it can be seen in Figures 30 and 31 that the use of Bayesian optimization seems to lead to a worse minimum compliance C, especially for a smaller mesh, with T having seemingly no influence.
An explanation for this can be seen in the Figures 32 and 33, where the likelihood of sampling good designs is low, especially in the case of $\mu = 1$. Bayesian Optimization, when geared towards exploitation, mainly searches the neighborhood of the best-found design points, and therefore relies on finding points close to the good compliance values. But as randomly sampling such points using just $M_0 = 200$ samples is unlikely (see Table 9), finding a good result using Bayesian optimization is rare. On the other hand, differential evolution uses 500000 samples and a particularly good underlying heuristic, allowing it to find better designs more reliably.
Increasing M_0 as well as the number of iterations might improve the overall result, but this will come at the cost of an increased computation time, possibly removing the advantage of faster computation over

differential evolution for all but the most expensive cost functions.
Therefore, the use of Bayesian optimization for optimizing over latent space would likely only make sense if the cost function were so expensive that the savings in computation time outweigh the less optimal designs found. In these cases, Bayesian optimization would enable latent space optimization, which would not be feasible using differential evolution or similar methods.
Nonetheless, it must be mentioned that Bayesian optimization is only possible during the third step of the proposed method, namely the optimization over latent space. But as the generation of training data also is time-consuming, Bayesian optimization during the latent space optimization would only bypass the problem of expensive cost functions if the local optimization operation LO would be significantly cheaper than the cost function evaluation c, or if training samples would already exist. This will be looked at in the next part.

Table 9: Values of the expected minimum compliance value E_{c_μ} of $M_0 = 200$ random samples for different designs, based on 500 runs.

$n_y \times n_x$	E_{c_1}	$E_{c_{10}}$	$E_{c_{100}}$
30×60	108.4	100.3	100.3
60×120	107.7	97.9	97.2

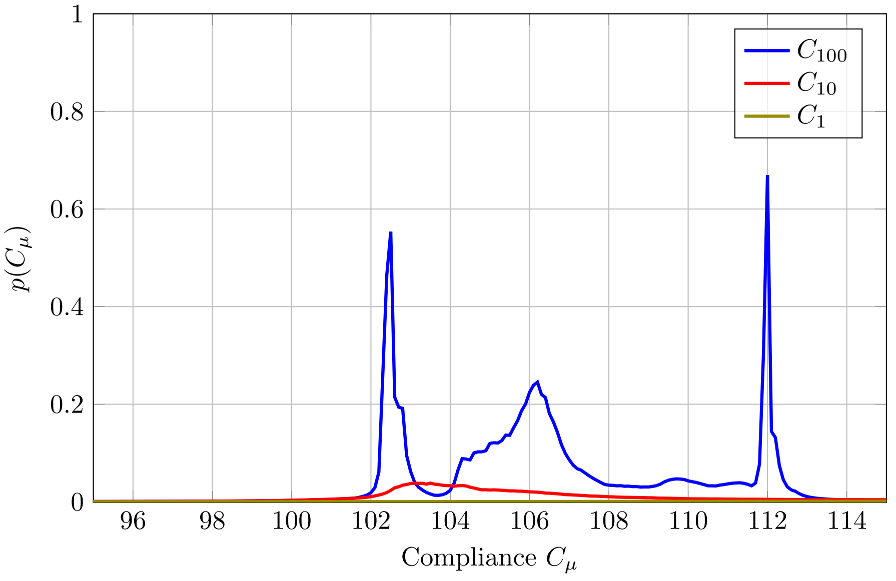


Figure 32: Probability density function of the compliance value C_μ, based on the k-nearest-neighbor method with 1000 neighbors from a set of 100000 samples [85]. The compliance values are based on the first network from section K.3, with $n_y \times n_x = 30 \times 60$.

K.4 Training one Network for varying boundary conditions

As the generation of the training data $\boldsymbol{X}_\lambda$ and the training of the decoder D on $\boldsymbol{X}_\lambda$ take up a significant part of the time used for this algorithm, it would be advantageous if different problems could be solved with the same decoder. To test this, different problems will be considered by varying the location $\boldsymbol{p}_F = (y_F, x_F)$ of the force F in Figure 18, while using the same Dirichlet boundary conditions (same bearings).
In a first step, three sets of training data $\boldsymbol{X}_{300}$ are generated according to the method seen in Appendix L.1. Two are created with $\boldsymbol{p}_{F,1} = (0, 0)$ and $\boldsymbol{p}_F = (15, 30)$ respectively, while for the third set, $\boldsymbol{p}_F$ is randomly chosen for every sample.
Thereafter, four decoders D for each set of training data (two possibilities for $m \in \{25, 50\}$ and two possibilities for using a signed distance field SDF or not) are trained, where pretraining for encoder and

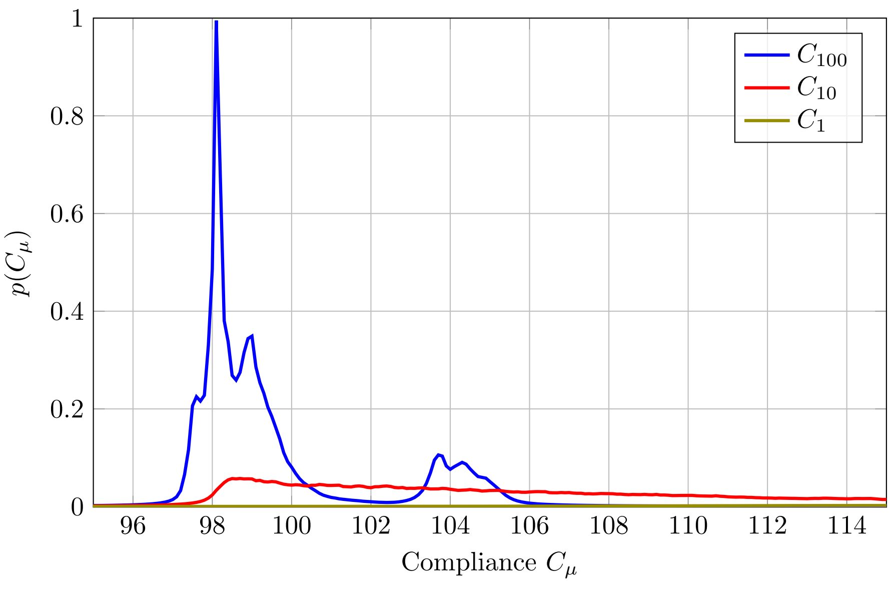


Figure 33: Probability density function of the compliance value C_μ, based on the k-nearest-neighbor method with 1000 neighbors from a set of 100000 samples [85]. The compliance values are based on the second network from section K.3, with $n_y \times n_x = 60 \times 120$.

decoder is used, as well as a discriminator network D_{Dis}, but not a surrogate model S (see Appendix L.2). After that, the decoders trained on the third training set with varying $\boldsymbol{p}_F$ are used to optimize the bridge problem from Figure 18 for $\boldsymbol{p}_{F,1}$ and $\boldsymbol{p}_{F,2}$ respectively. In each case (like in Appendix K.1) this is done with $\mu = 1$ as well as $\mu = 10$ and a subsequent post-processing, but also by the random initialization of 500 samples in latent space with a following post-processing.
Similarly, the decoders trained on the first two training sets are then used to optimize the problem (with the respective $\boldsymbol{p}_{F,i}$) to allow a comparison between the different approaches.
Equivalently to sections K.1 and K.2, every decoder is trained twice for every case, to limit the influence of randomness onto the results.

Results

In the Figures 34 and 35, it can be seen that in nearly every case of using a neural network which has been trained for the force being applied in the specific position $\boldsymbol{p}_F$, one will get a result, which after post-processing is better than the result C_h one gets when using only SIMP with an homogeneous input. But on the other hand, using a network trained for all possible $\boldsymbol{p}_F$ will in all but one case ($\boldsymbol{p}_F = (15, 30)$, $m = 25$, with SDF) not be able to surpass the homogeneous solution C_h.
Consequently, the possibility of a network trained on a broad set of problems being able to solve every possible problem seems unlikely. If computation time is an important factor, and training samples cannot be procured from somewhere else, this hampers the viability of the proposed method at least for compliance minimization, as the expensive generation of training data and training of the autoencoder would be required for every single optimization.

K.5 On the usefullnes of the cost function

Besides the other problems with the proposed method outlined earlier, like the expensive generation of training data, there is another problem worth discussing. Namely, the difficulty of constructing a cost function which allows for fair comparisons between different designs.
Firstly, it has to be mentioned that every design $\boldsymbol{x}$ considered is not manufacturable, due to amount of corners introduced by the mesh and the difficulty of transferring grey density values $x_i \in (0, 1)$ into a real designs. Consequently, $\boldsymbol{x}$ has to be considered as only a representation of the true design $\boldsymbol{\Phi}$, which accurately might be represented as a level set function. Therefore, the compliance values $C(\boldsymbol{x})$ used in this work are likely also only approximations of the true compliance value $C_{\boldsymbol{\Phi}}$. As there are multiple

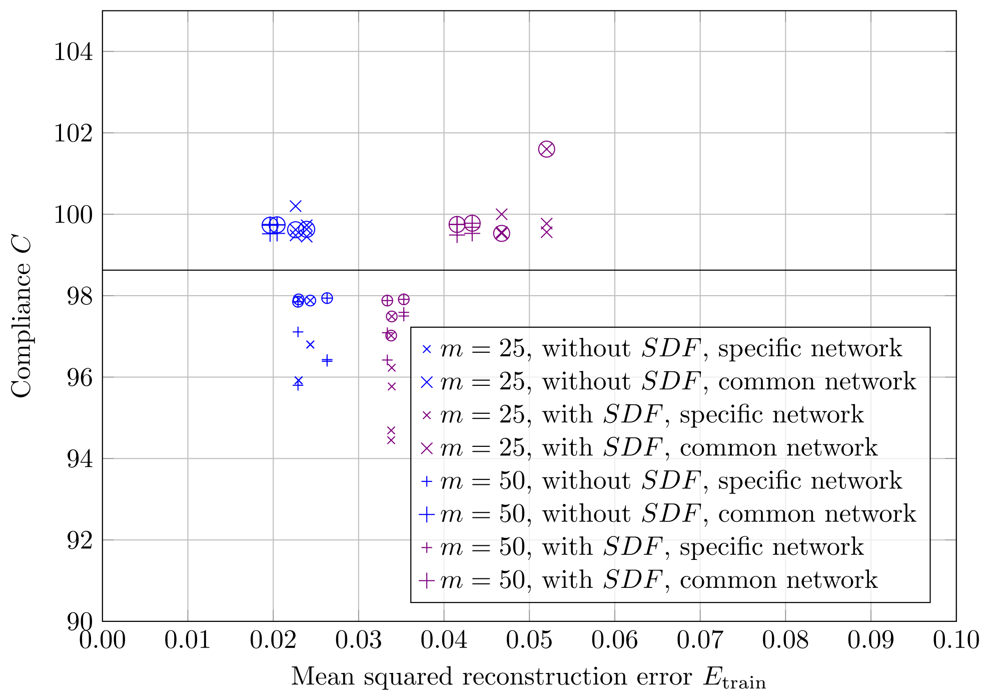


Figure 34: Reconstruction error for different network setups, with the force attacking at node $\boldsymbol{p}_F = (0, 0)$. Encircled data points correspond to results found by post-processing random latent space samples. The black line corresponds to the compliance value found after 1000 steps of SIMP with a homogeneous input ($C \approx 98.627$).

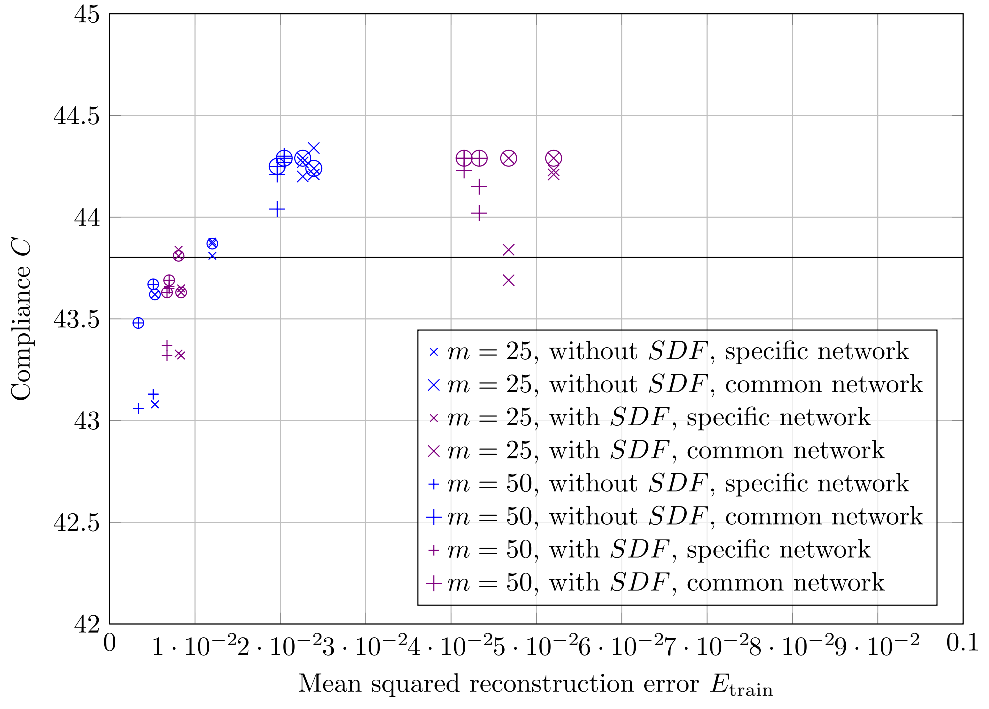
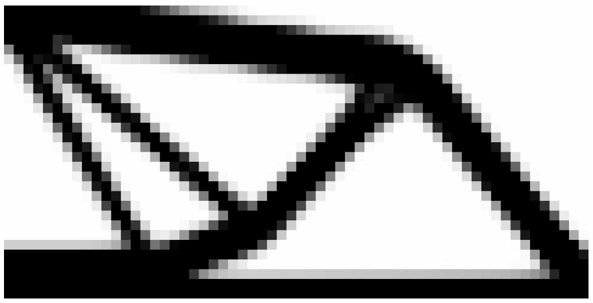
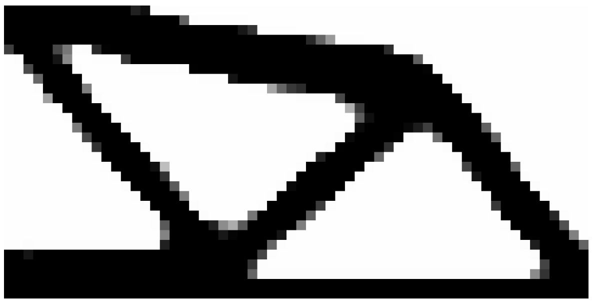
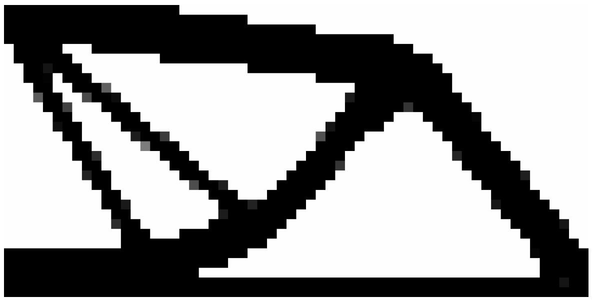


Figure 35: Reconstruction error for different network setups, with the force attacking at node $\boldsymbol{p}_F = (15, 30)$. Encircled data points correspond to results found by post-processing random latent space samples. The black line corresponds to the compliance value found after 1000 steps of SIMP with a homogeneous input ($C \approx 43.803$).

different approximations $\boldsymbol{x}$ for $\boldsymbol{\Phi}$ (for example by using grey values or not), multiple different $C(\boldsymbol{x})$ can be connected to each $\boldsymbol{\Phi}$. This can be seen for example in the Figures 36, 38, and 39, where different design $\boldsymbol{x}$ can be used to approximate the same topology $\boldsymbol{\Phi}$.
The main problem now is that the relation between the compliance values $C(\boldsymbol{x}_1)$ and $C(\boldsymbol{x}_2)$ of two topologies $\boldsymbol{\Phi}_1$ and $\boldsymbol{\Phi}_2$ can change depending on the approximation method used. Therefore, inferring the relation of $C_{\boldsymbol{\Phi}_1}$ and $C_{\boldsymbol{\Phi}_2}$ from $C(\boldsymbol{x}_1)$ and $C(\boldsymbol{x}_2)$ becomes impossible. This makes most comparisons between the approximated compliance values meaningless.
And while such comparisons are not necessarily required for the first step of the proposed method, the generation of training data (here, using approximations should be fine), the optimization over latent space and the post-processing would be significantly hindered, as they rely on comparisons between different designs.
A possible solution here would be a transformation T ($\boldsymbol{\Phi} = T(\boldsymbol{x})$) to get the true topology, whose compliance $C_{\boldsymbol{\Phi}}$ could then be calculated, for example using the enriched FEM method [86].
For the local optimization of the topology $\boldsymbol{\Phi}$, which are required for the last two steps of the proposed method, non gradient-based methods like the Nelder-Mead algorithm [57] can be used if no gradient is available.


(a) Design after 1000 steps of SIMP
$C \approx 98.63$


(b) Best design after post-processing
$C \approx 94.03$


(c) Design after 1000 steps of SIMP, filtered
$C \approx 92.01$

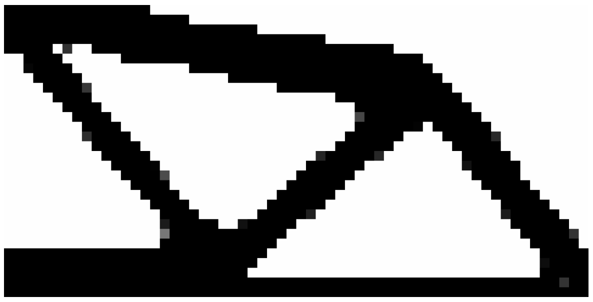

(d) Best design after post-processing, filtered
$C \approx 92.54$

Figure 36: Different designs for $n_y \times n_x = 30 \times 60$.

K.6 Discussion

Overall, the proposed method for overcoming the convexity in topology optimization problems by reducing the dimensionality using autoencoders and using Bayesian optimization to shorten the optimization over latent space shows promise in that it is able to produce results with a significantly lower cost function value. Especially the use of signed distance functions during network training and optimization over latent space often allows for the creation of stiffer designs when trying to minimize compliance. But beside some promising results, the whole process is still beset by several problems.

- The generation of the training data takes a lot of time, and scales linearly with the cost function expense (compared for example to the neural network, the training time of which is not dependent on the cost function). This problem might be avoided by training a neural network which predicts the optimized design from the initial material distribution in a single step for different boundary conditions. However, it is not clear if training such a network is feasible, or how this method for generating training samples could affect the overall process.

- No single tested setup can reliably lead to results which are better than the best design of the training set.
- When using Bayesian optimization, the compliance values get even worse, making the production of a superior design even less likely.
- The whole process is based on the assumption that the compliance of different topologies can be compared fairly, something that in the current implementation of the method is likely not the case. Avoiding this problem would likely result in a large computational expense.

Consequently, the proposed method has the potential to overcome the barriers preventing a global optimization for larger topology optimization problems, but significant improvements are required to overcome the current problems.
In regard to this work, it has to be noted that the tested problem of compliance minimization is fairly well behaved, with the standard approach of SIMP with homogeneous inputs leading to fairly usefull results. Therefore, the method might be more helpfull in problems which suffer far more from the issue of non-convexity and exploring such problems might be worth further research.

L Implementation of the problem in Appendix K.1

In this part, a detailed description of the process used to optimize the topology designs will be given.

L.1 Generating the training set

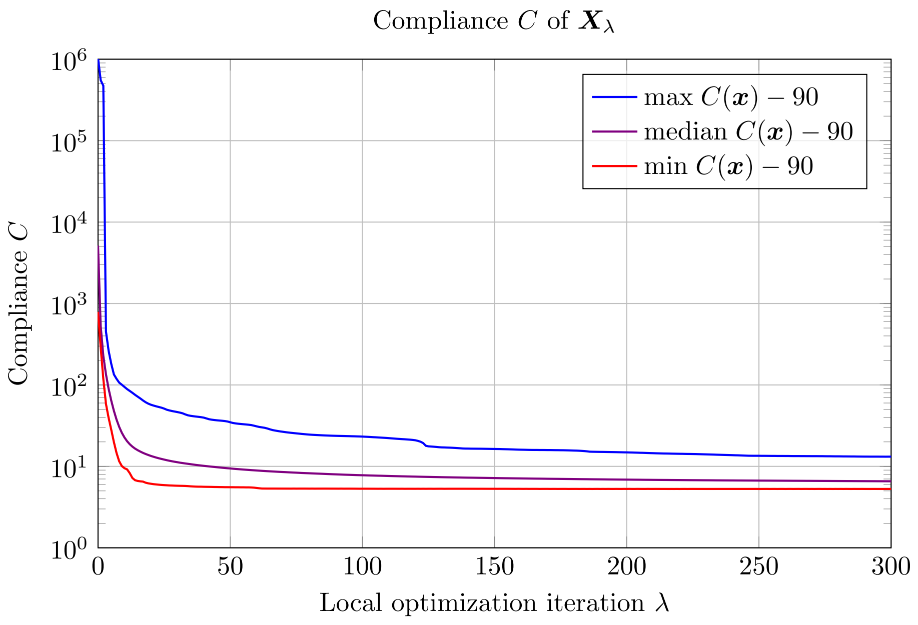


Figure 37: The compliance of 20.000 randomly initiated samples after λ iterations, with $n_y = 45$ and $n_x = 90$.

The first step in the process here is the creation of the training data, with $N = 25000$ samples being created for each test case of n_y, n_x, and λ, with this being overall nine cases. $\lambda = 0$ will be looked at first, as it is the case with random variables, which would give a benchmark for comparison. Therefore, it would be advantageous that the created samples fulfill the volume constraint $\overline{\boldsymbol{x}} \overset{!}{=} \rho_0 = 0.4$. To achieve this, in a first step $\widetilde{\boldsymbol{x}}$ is sampled randomly, with each element $\widetilde{x}_i$ being sampled according to the following probability density function $p(\widetilde{x}_i)$:

$$p(\widetilde{x}_i) = \begin{cases} 1 & 0 \leq \widetilde{x}_i \leq 1 \\ 0 & \text{else} \end{cases} \tag{91}$$

But as this would likely lead to $\overline{\widetilde{\boldsymbol{x}}} \approx 0.5$, further steps have to be taken. In this case, one would transform each element using $x_i = g(\widetilde{x}_i) = \widetilde{x}_i^a$ with $a > 0$. This specific transformation is chosen, as it preserves the boundaries of $0 \leq x_i, \widetilde{x}_i \leq 1$. a now has to be chosen so that $\overline{\boldsymbol{x}} \overset{!}{=} \rho_0$:

$$\begin{aligned} \mathbb{E}(g(\widetilde{x}_i)) &= \int_{-\infty}^{\infty} g(\widetilde{x}_i)p(\widetilde{x}_i)\mathrm{d}\widetilde{x}_i \\ \overline{\boldsymbol{x}} &\approx \int_0^1 \widetilde{x}_i^a \mathrm{d}\widetilde{x}_i = \frac{1}{a+1} \overset{!}{=} \rho_0 \end{aligned} \tag{92}$$

From this, it now follows that

$$a \overset{!}{=} \frac{1}{\rho_0} - 1 = \frac{3}{2} \tag{93}$$

After generating the training set $\boldsymbol{X}_0$ in such a way, one than uses the basic SIMP algorithm [56] for local optimization, although it was improved upon by using a more refined method in assembling the stiffness matrix [66], leading to significantly faster computation times. During each step, filtering was used, with a filter radius $r_{\min} = 1.25$ (see Appendix A.2).
This method was then used to create the training sets $\boldsymbol{X}_\lambda$ for the autoencoder. $\lambda = 30$ was chosen, as it would give an intermediate case, where the training set already include very good design (close to the optimum after $\lambda = 300$), while still including designs which can be improved greatly. This means that there is a greater variance in the training set. Lastly, $\lambda = 300$ is selected, as all design at this stage seem to arrive close to their respective local minimum, giving a training set with designs as good as possible (see Figure 37).
In each case for $n_y \times n_x$ and λ, a randomly selected fifth of the training data (5000 samples) will be removed from the training set and will later be used for validation.

L.2 Training of the autoencoder

After generating a training set in the first step, a autoencoder (see Appendix A.6) has to be trained on it.
In this part, $\boldsymbol{z}$ in this work will always be a one-dimensional vector, with the latent space $Z = [0,1]^m$ and $m = \nu^{(l_B)}$. Furthermore, in every autoencoder used for compliance minimization, the input layer and output layer will be three dimensional tensors, but with $\kappa^{(0)} = \kappa^{(L)} = 1$. Consequently, $\boldsymbol{x}$ and $\widehat{\boldsymbol{x}}$ will be treated as being two-dimensional, with $\boldsymbol{x}, \widehat{\boldsymbol{x}} \in X = [0,1]^{n_y \times n_x}$, with $n_y n_x = n$, and X being the design domain.
In the following, the parameters used for this step shall be presented. First to mention here are the weights $\boldsymbol{w}_i$ from the reconstruction loss function (see equation (45)), which when using a signed distance field, are:

$$[w_i]_{k,l} = \begin{cases} 1 & \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 1 \\ 0.5 & 1 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 2 \\ 0.25 & 2 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 5 \\ 0.1 & 5 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 10 \\ 0.025 & 10 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \leq 20 \\ 0.01 & 20 < \left|[SDF(\boldsymbol{x}_i)]_{k,l}\right| \end{cases} \tag{94}$$

Meanwhile, if the density field itself was encoded instead, the following weight matrix was used:

$$[w_i]_{k,l} = 2\left(1 - \left|\,[x_i]_{k,l} - 0.5\right|\right) \tag{95}$$

Meanwhile, when a discriminator network D_{Dis} is used, it tries to enforce a uniform distribution over the bounded latent space, with $P_{\text{Dis}} = \frac{1}{4}$.
In the case of the surrogate network being used, one then will set $\beta_S = \frac{1}{4}$.
And finally, in the case of pretraining, encoder and decoder are split in two, and just like in the Appendix I.2, with each layer being trained for 50 epochs, with the first 45 epochs having 10 batches each, while the last 5 epochs use only one batch each. The exact encoder and decoder networks used can bee seen in Figures 41, 42, and 43, the specific networks used for discriminator D_{Dis} and surrogate model S in Figure 44.

L.3 Optimization over latent space

After a Decoder $D : \mathbb{R}^m \to \mathbb{R}^n$ has been build, for each of them, optimization over the latent space $Z = [0, 1]^m$ will be performed.
While Appendix K.1 suggest the use of the cost function $c_\mu(\boldsymbol{z})$ from equation (3), in this case, a more broader function is used, due to the fact that the local optimization algorithm LO (SIMP) can worsen the compliance of a design ($c(LO^{i+1}(\boldsymbol{x}) > c(LO^i(\boldsymbol{x}))$ is possible). Therefore, the following cost function will be optimized instead:

$$c_\mu(\boldsymbol{z}) = \min_{i \in \{1,\dots,\mu\}} \left\{ c\left(LO^i\left(D(\boldsymbol{z})\right)\right) : \overline{LO^i\left(D(\boldsymbol{z})\right)} \leq \rho_0 \right\} \tag{96}$$

It has to be noted that, especially in the case of $\mu = 1$, the set from which the minimum will be chosen might be empty, as there could be no element which fulfills the mass constraint. In this case, μ will be increased until the first viable design has been produced. It has to be noted that during the test, this was always achieved after a maximum of two steps, so $\mu = 2$ was enough to find feasible designs in every test done. For each decoder, two optimizations where performed, once with $\mu = 1$, as well as with $\mu = 10$.
In this work, the authors use differential evolution $DE_{5m,500,0.6,0.9}$ to generate $\boldsymbol{P}_{500}$, the final result of the optimization process.
Additionally, to get an estimation on the effectiveness of step 3, the global optimization using differential evolution, one will also use a random local approach. In this, $20m$ random points are chosen, sampled according to a uniform distribution over Z. These points are then directly post-processed, again for 500 steps. The best design found doing this can then be compared to the $\boldsymbol{x}_{\min}$ found for $\mu \in \{1, 10\}$.

L.4 Post-processing

After optimization over latent space, post-processing will be done. While it would be the simplest approach to just take the best latent space design (with the cost function from equation (96))

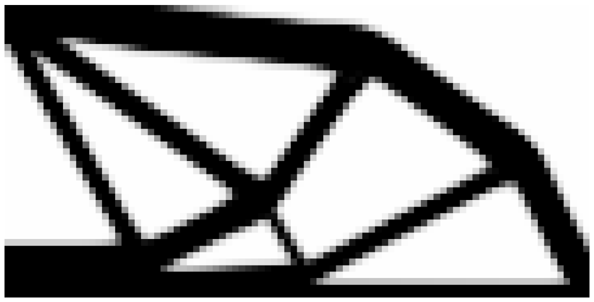
$$\boldsymbol{z}^* = \underset{\boldsymbol{z} \in \boldsymbol{P}_{500}}{\text{argmin}} \{c_\mu(\boldsymbol{z})\}, \tag{97}$$

one instead will use post-processing for all designs included in $\boldsymbol{P}_{500}$. The main reason here is that it is possible that $\boldsymbol{P}_{500}$ will include multiple optima with nearly equivalent cost function value. As it is possible that post-production will make a inferior design in latent space superior in the design space after a number of local optimizations, this would increase the likelihood of finding the global optimum. Additionally, it is possible to parallelize the post-processing of multiple designs with minimal overhead, which reduces the additional computation time needed for post-processing every member of $\boldsymbol{P}_{500}$ (it is possible to parallelize the post-processing of a single design, but in an environment with distributed memory, this would include a far larger overhead).
The final design $\boldsymbol{x}_{i,f} \in \boldsymbol{X}_f$ of a latetnt space design $\boldsymbol{z}_i = \boldsymbol{p}_{i,500} \in \boldsymbol{P}_{500}$ can then be found in the following way, with 500 steps of post-processing being used in this work:

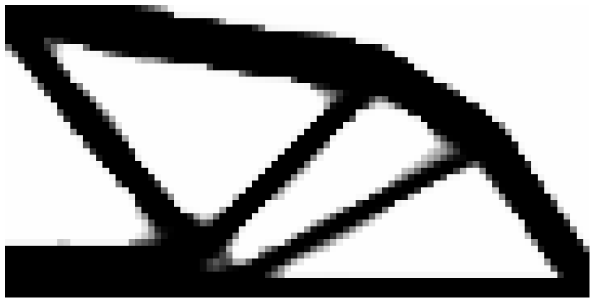
$$\boldsymbol{x}_{i,f} = LO^{j_{\min}}\left(D(\boldsymbol{z}_i)\right) = \underset{j \in \{1,\dots,500\}}{\text{argmin}} \left\{ c\left(LO^j\left(D(\boldsymbol{z}_i)\right)\right) : \overline{LO^j\left(D(\boldsymbol{z}_i)\right)} \leq \rho_0 \right\} \tag{98}$$

The final result $\boldsymbol{x}_{\min}$ found by the whole process will then be the $\boldsymbol{x}_{i,f}$ which has the minimal cost function value:

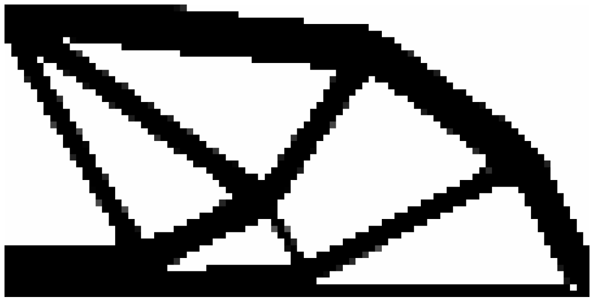
$$\boldsymbol{x}_{\min} = \underset{\boldsymbol{x}_{i,f} \in \boldsymbol{X}_f}{\text{argmin}} \{c(\boldsymbol{x}_{i,f})\} \tag{99}$$



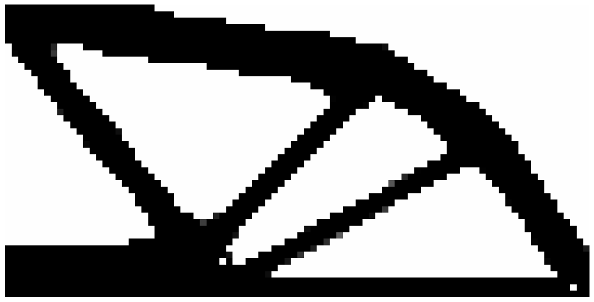
(a) Design after 1000 steps of SIMP
$C \approx 95.42$



(b) Best design after post-processing
$C \approx 94.33$

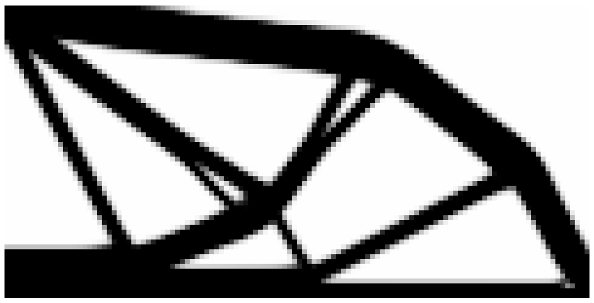


(c) Design after 1000 steps of SIMP, filtered
$C \approx 90.50$

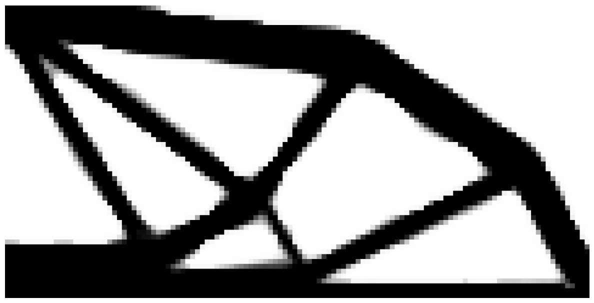


(d) Best design after post-processing, filtered
$C \approx 91.87$

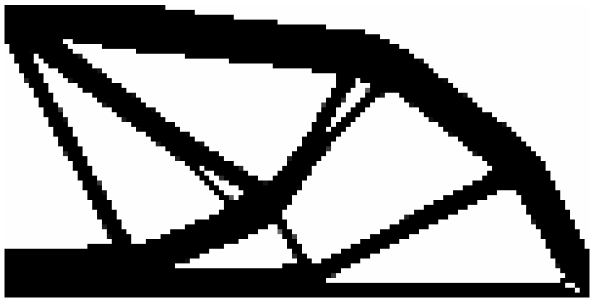
Figure 38: Different designs for $n_y \times n_x = 45 \times 90$.



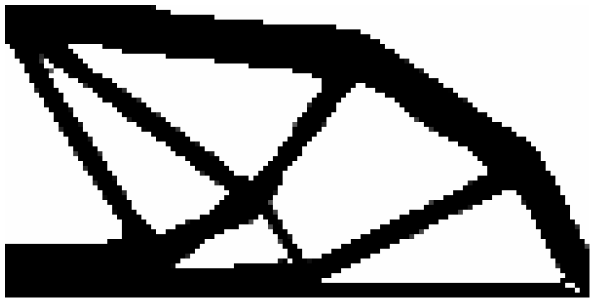
(a) Design after 1000 steps of SIMP
$C \approx 94.54$



(b) Best design after post-processing
$C \approx 93.67$

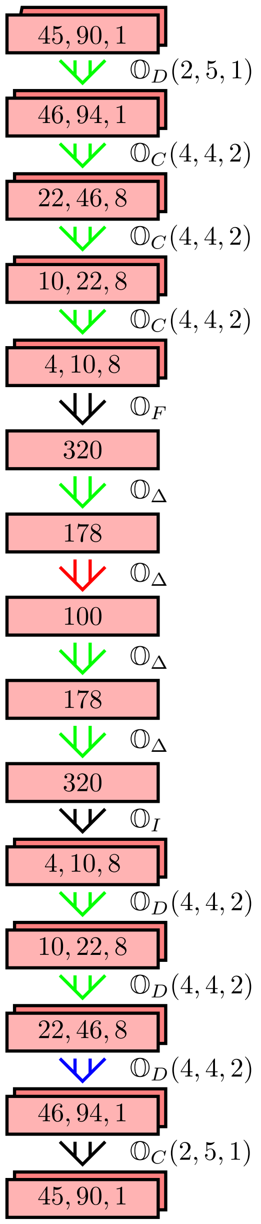


(c) Design after 1000 steps of SIMP, filtered
$C \approx 90.77$

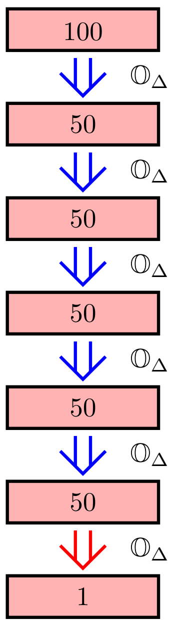


(d) Best design after post-processing, filtered
$C \approx 91.11$

Figure 39: Different designs for $n_y \times n_x = 60 \times 120$.

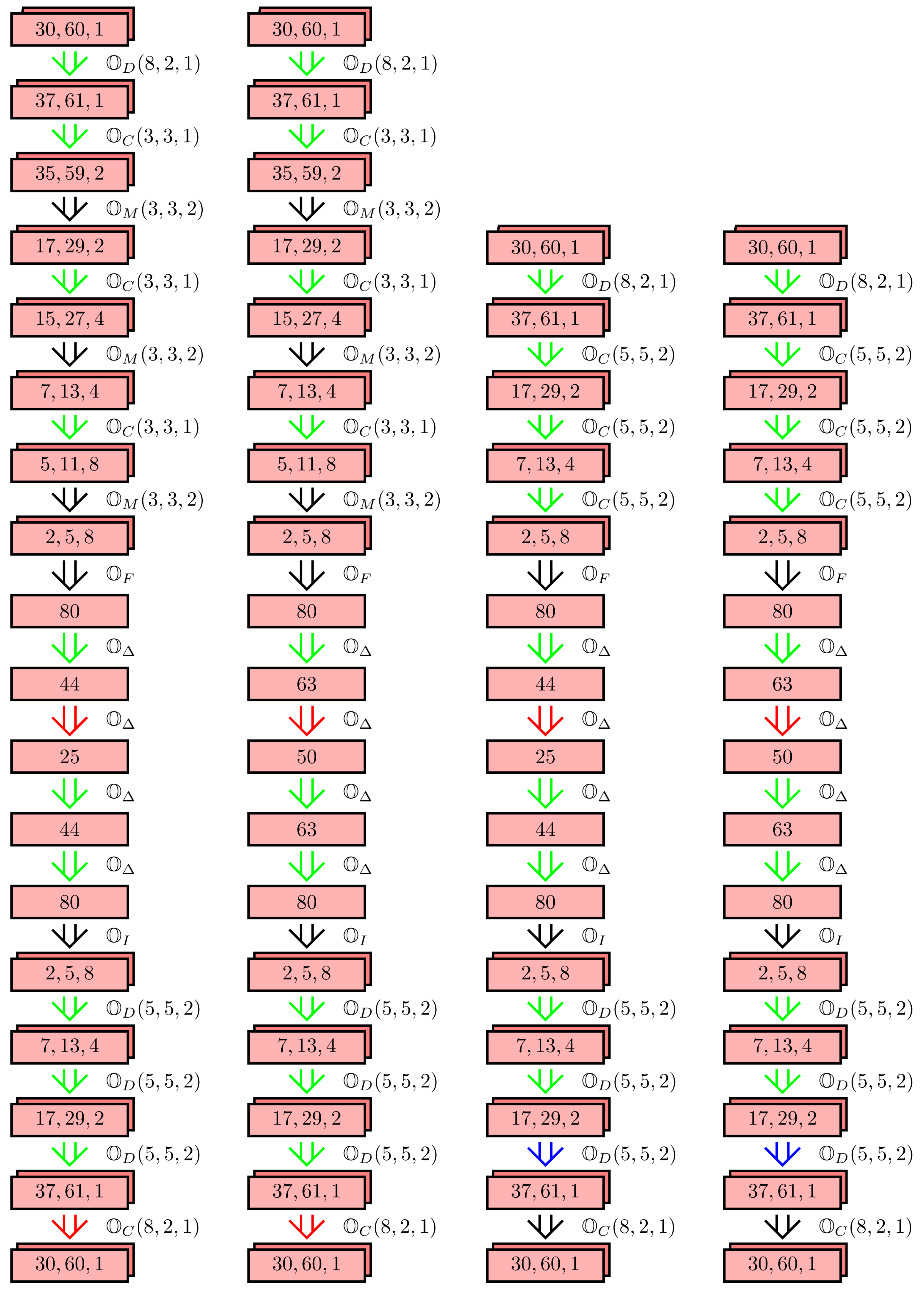


(a) Combination of encoder E and decoder D.



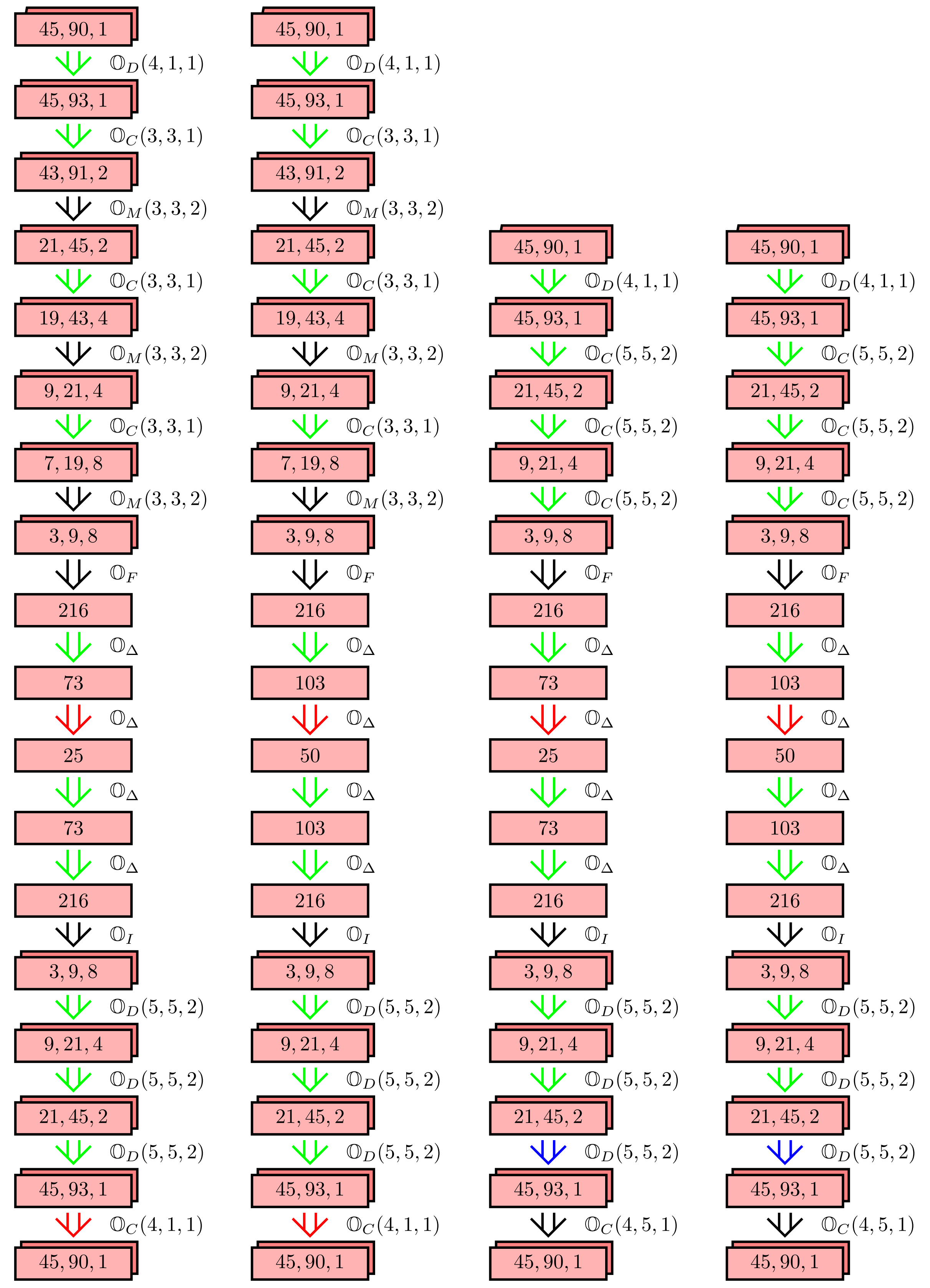
(b) Discriminator D_{Dis} and surrogate model S.

Figure 40: Neural networks used in section 1.4 (see Figure 7 for symbol explanation). Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).



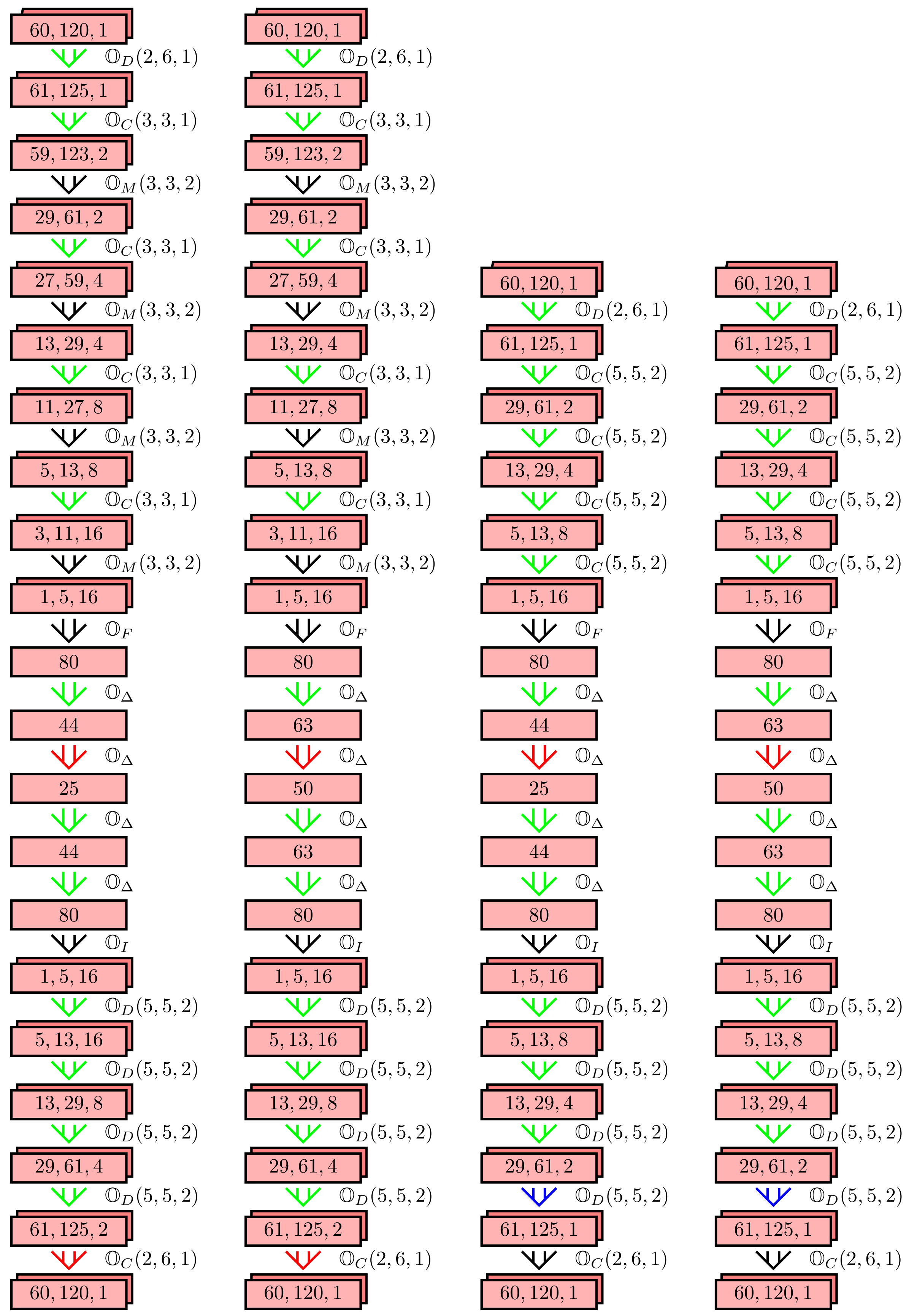
(a) $m = 25$, without SDF (b) $m = 50$, without SDF (c) $m = 25$, with SDF (d) $m = 50$, with SDF

Figure 41: The encoder E and decoder D networks used for $n_y \times n_x = 30 \times 60$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).



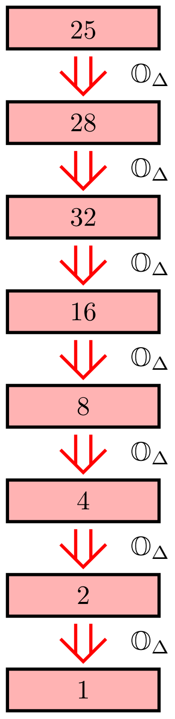
(a) $m = 25$, without SDF (b) $m = 50$, without SDF (c) $m = 25$, with SDF (d) $m = 50$, with SDF

Figure 42: The encoder E and decoder D networks used for $n_y \times n_x = 45 \times 90$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).

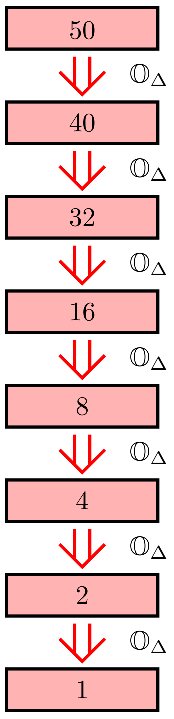


(a) $m = 25$, without SDF (b) $m = 50$, without SDF (c) $m = 25$, with SDF (d) $m = 50$, with SDF

Figure 43: The encoder E and decoder D networks used for $n_y \times n_x = 60 \times 120$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).

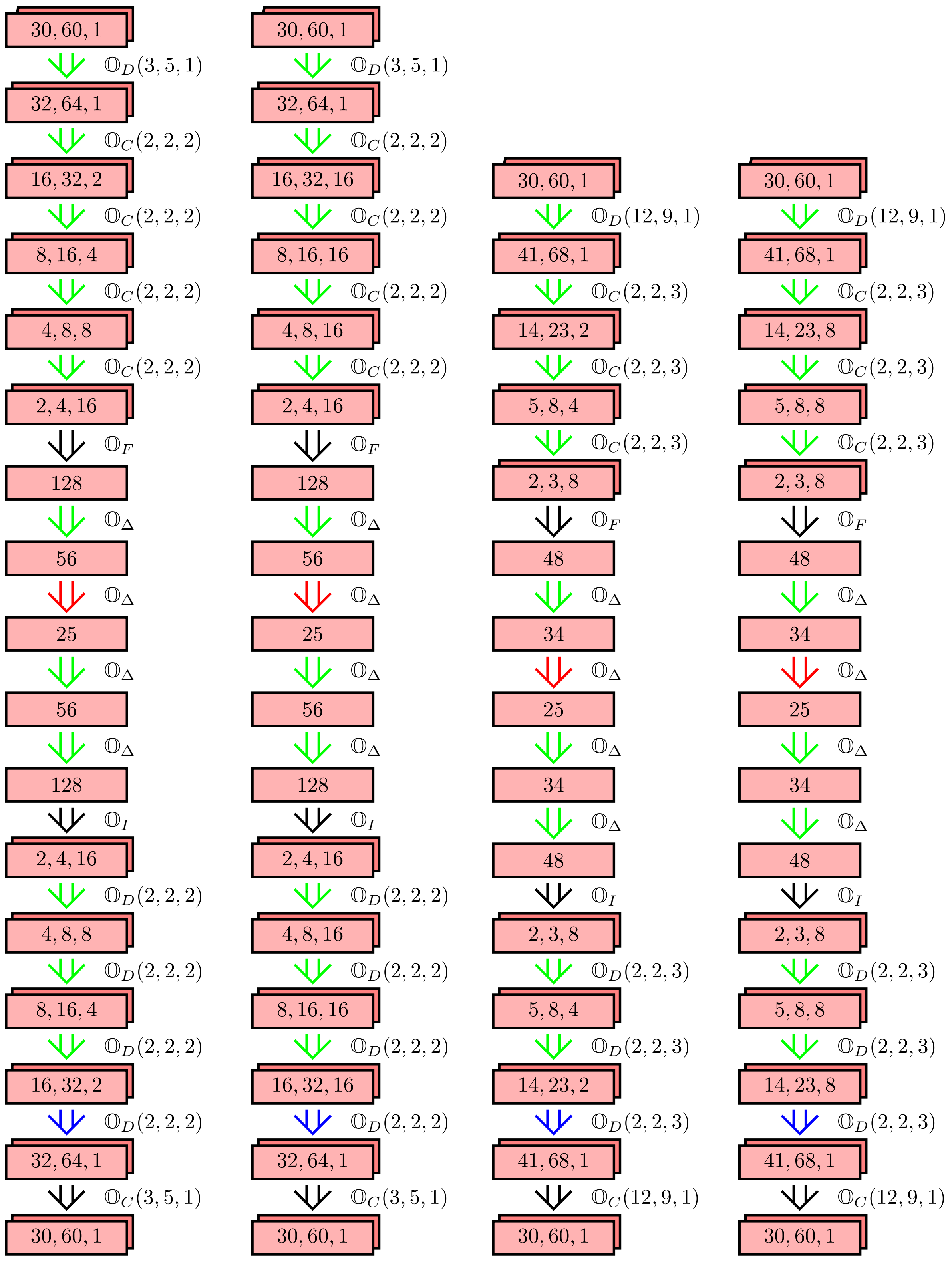


(a) $m = 25$



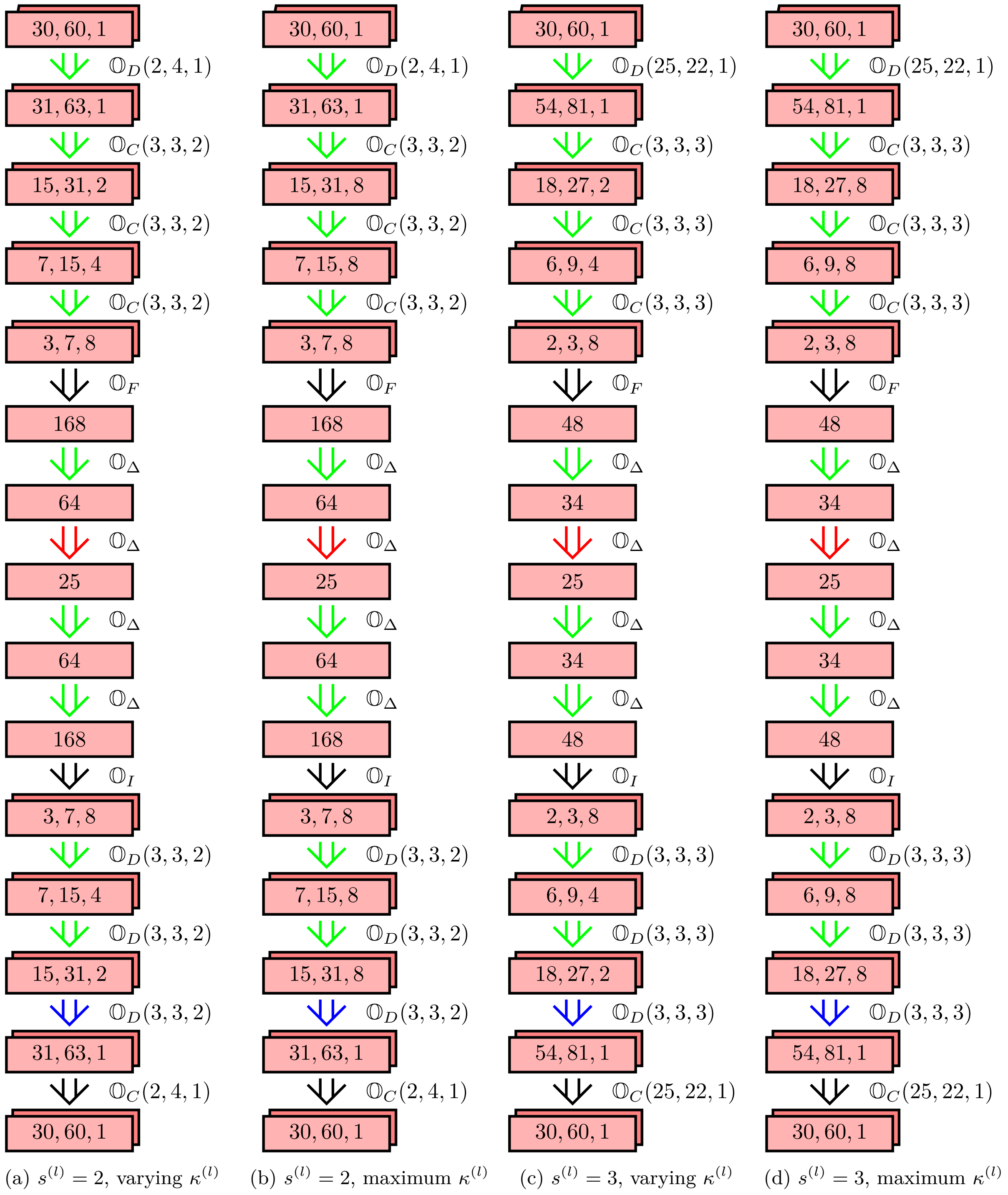
(b) $m = 50$

Figure 44: The discrimiator D_{Dis} and surrogate model S networks. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).



(a) $s^{(l)} = 2$, varying $\kappa^{(l)}$ (b) $s^{(l)} = 2$, maximum $\kappa^{(l)}$ (c) $s^{(l)} = 3$, varying $\kappa^{(l)}$ (d) $s^{(l)} = 3$, maximum $\kappa^{(l)}$

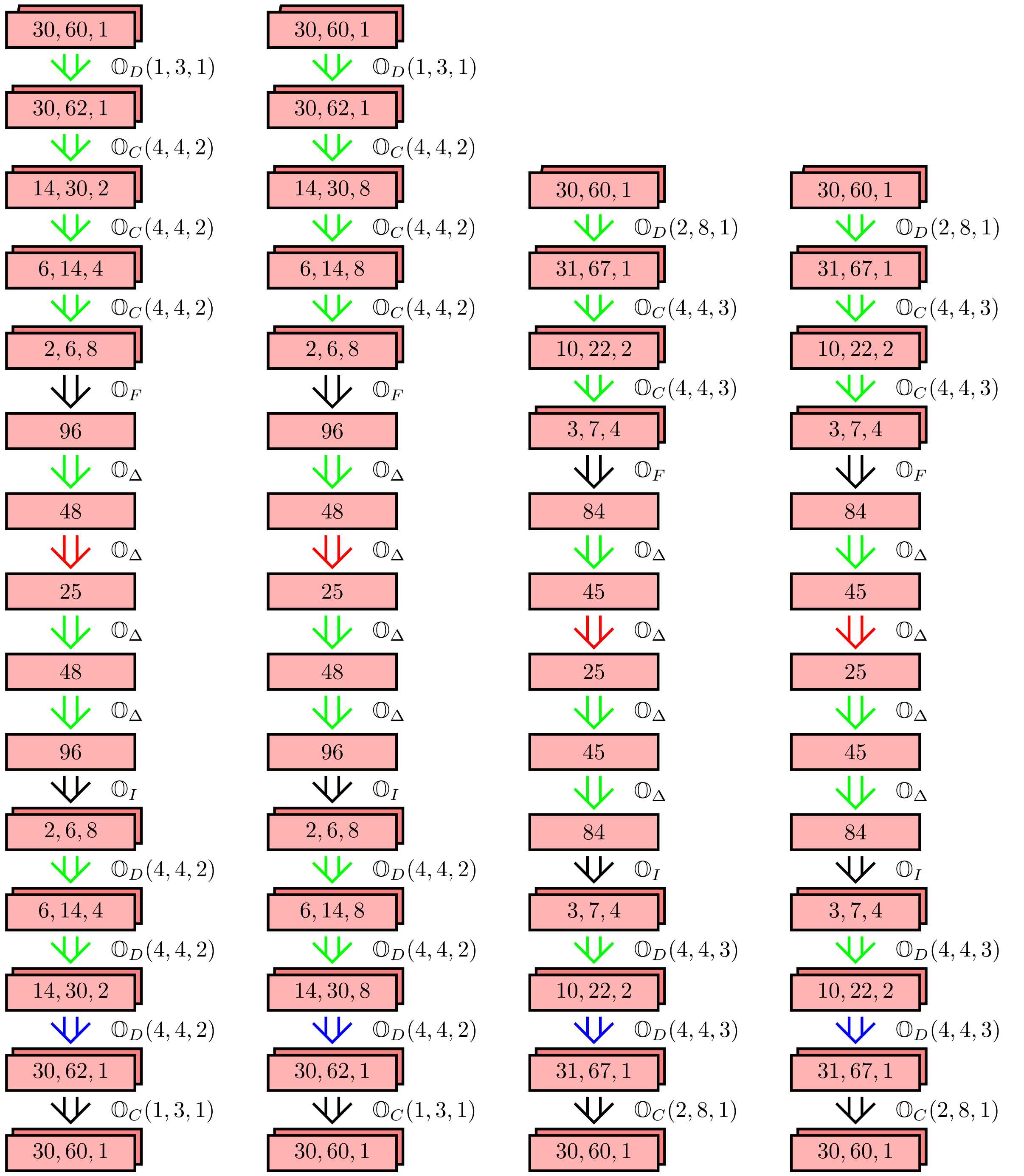
Figure 45: The encoder E and decoder D networks used for $t^{(l)} = 2$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).



(a) $s^{(l)} = 2$, varying $\kappa^{(l)}$: $30,60,1 \xrightarrow{\mathbb{O}_D(2,4,1)} 31,63,1 \xrightarrow{\mathbb{O}_C(3,3,2)} 15,31,2 \xrightarrow{\mathbb{O}_C(3,3,2)} 7,15,4 \xrightarrow{\mathbb{O}_C(3,3,2)} 3,7,8 \xrightarrow{\mathbb{O}_F} 168 \xrightarrow{\mathbb{O}_\Delta} 64 \xrightarrow{\mathbb{O}_\Delta} 25 \xrightarrow{\mathbb{O}_\Delta} 64 \xrightarrow{\mathbb{O}_\Delta} 168 \xrightarrow{\mathbb{O}_I} 3,7,8 \xrightarrow{\mathbb{O}_D(3,3,2)} 7,15,4 \xrightarrow{\mathbb{O}_D(3,3,2)} 15,31,2 \xrightarrow{\mathbb{O}_D(3,3,2)} 31,63,1 \xrightarrow{\mathbb{O}_C(2,4,1)} 30,60,1$
(b) $s^{(l)} = 2$, maximum $\kappa^{(l)}$: $30,60,1 \xrightarrow{\mathbb{O}_D(2,4,1)} 31,63,1 \xrightarrow{\mathbb{O}_C(3,3,2)} 15,31,8 \xrightarrow{\mathbb{O}_C(3,3,2)} 7,15,8 \xrightarrow{\mathbb{O}_C(3,3,2)} 3,7,8 \xrightarrow{\mathbb{O}_F} 168 \xrightarrow{\mathbb{O}_\Delta} 64 \xrightarrow{\mathbb{O}_\Delta} 25 \xrightarrow{\mathbb{O}_\Delta} 64 \xrightarrow{\mathbb{O}_\Delta} 168 \xrightarrow{\mathbb{O}_I} 3,7,8 \xrightarrow{\mathbb{O}_D(3,3,2)} 7,15,8 \xrightarrow{\mathbb{O}_D(3,3,2)} 15,31,8 \xrightarrow{\mathbb{O}_D(3,3,2)} 31,63,1 \xrightarrow{\mathbb{O}_C(2,4,1)} 30,60,1$
(c) $s^{(l)} = 3$, varying $\kappa^{(l)}$: $30,60,1 \xrightarrow{\mathbb{O}_D(25,22,1)} 54,81,1 \xrightarrow{\mathbb{O}_C(3,3,3)} 18,27,2 \xrightarrow{\mathbb{O}_C(3,3,3)} 6,9,4 \xrightarrow{\mathbb{O}_C(3,3,3)} 2,3,8 \xrightarrow{\mathbb{O}_F} 48 \xrightarrow{\mathbb{O}_\Delta} 34 \xrightarrow{\mathbb{O}_\Delta} 25 \xrightarrow{\mathbb{O}_\Delta} 34 \xrightarrow{\mathbb{O}_\Delta} 48 \xrightarrow{\mathbb{O}_I} 2,3,8 \xrightarrow{\mathbb{O}_D(3,3,3)} 6,9,4 \xrightarrow{\mathbb{O}_D(3,3,3)} 18,27,2 \xrightarrow{\mathbb{O}_D(3,3,3)} 54,81,1 \xrightarrow{\mathbb{O}_C(25,22,1)} 30,60,1$
(d) $s^{(l)} = 3$, maximum $\kappa^{(l)}$: $30,60,1 \xrightarrow{\mathbb{O}_D(25,22,1)} 54,81,1 \xrightarrow{\mathbb{O}_C(3,3,3)} 18,27,8 \xrightarrow{\mathbb{O}_C(3,3,3)} 6,9,8 \xrightarrow{\mathbb{O}_C(3,3,3)} 2,3,8 \xrightarrow{\mathbb{O}_F} 48 \xrightarrow{\mathbb{O}_\Delta} 34 \xrightarrow{\mathbb{O}_\Delta} 25 \xrightarrow{\mathbb{O}_\Delta} 34 \xrightarrow{\mathbb{O}_\Delta} 48 \xrightarrow{\mathbb{O}_I} 2,3,8 \xrightarrow{\mathbb{O}_D(3,3,3)} 6,9,8 \xrightarrow{\mathbb{O}_D(3,3,3)} 18,27,8 \xrightarrow{\mathbb{O}_D(3,3,3)} 54,81,1 \xrightarrow{\mathbb{O}_C(25,22,1)} 30,60,1$

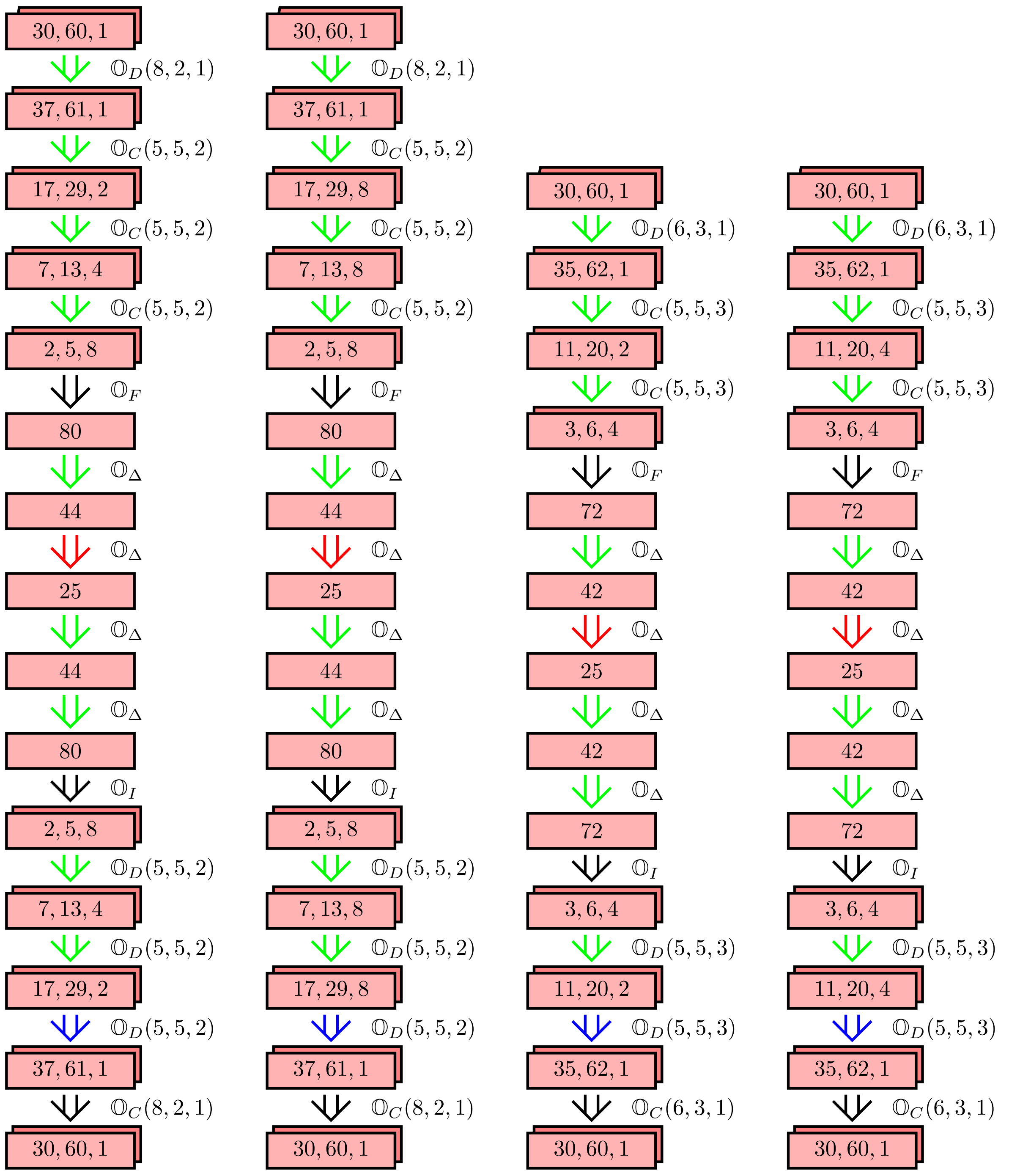
(a) $s^{(l)} = 2$, varying $\kappa^{(l)}$ (b) $s^{(l)} = 2$, maximum $\kappa^{(l)}$ (c) $s^{(l)} = 3$, varying $\kappa^{(l)}$ (d) $s^{(l)} = 3$, maximum $\kappa^{(l)}$

Figure 46: The encoder E and decoder D networks used for $t^{(l)} = 3$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).



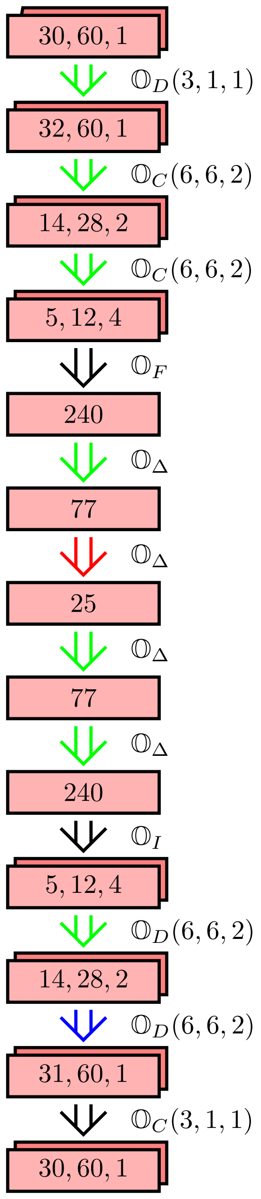
(a) $s^{(l)} = 2$, varying $\kappa^{(l)}$ (b) $s^{(l)} = 2$, maximum $\kappa^{(l)}$ (c) $s^{(l)} = 3$, varying $\kappa^{(l)}$ (d) $s^{(l)} = 3$, maximum $\kappa^{(l)}$

Figure 47: The encoder E and decoder D networks used for $t^{(l)} = 4$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).

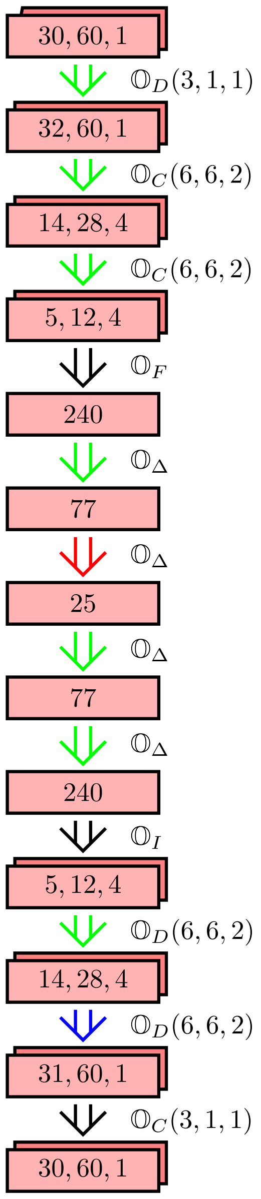


(a) $s^{(l)} = 2$, varying $\kappa^{(l)}$ (b) $s^{(l)} = 2$, maximum $\kappa^{(l)}$ (c) $s^{(l)} = 3$, varying $\kappa^{(l)}$ (d) $s^{(l)} = 3$, maximum $\kappa^{(l)}$

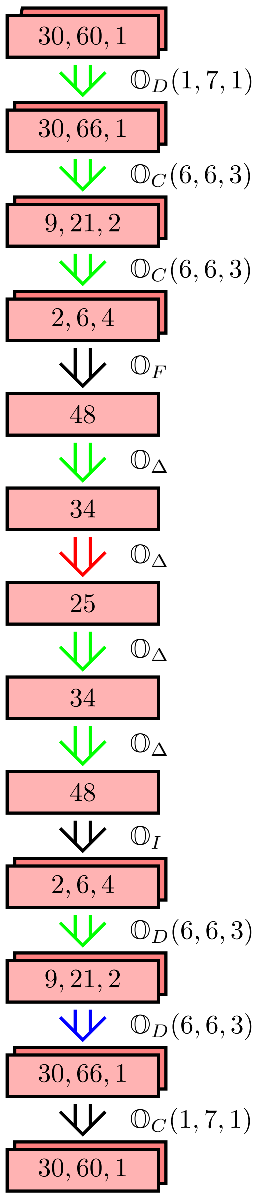
Figure 48: The encoder E and decoder D networks used for $t^{(l)} = 5$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).



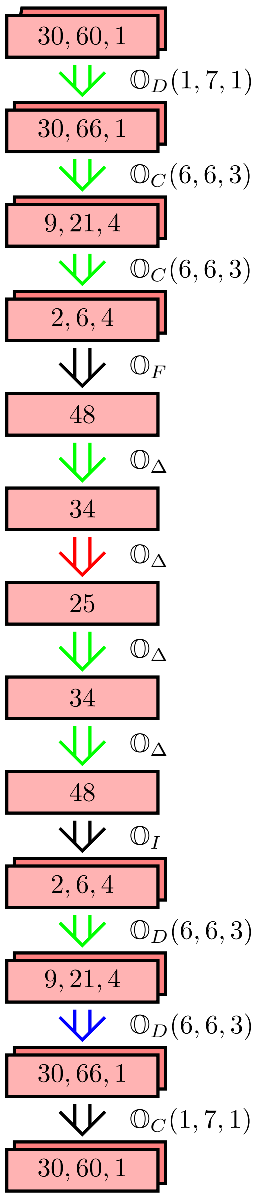
(a) $s^{(l)} = 2$, varying $\kappa^{(l)}$



(b) $s^{(l)} = 2$, maximum $\kappa^{(l)}$



(c) $s^{(l)} = 3$, varying $\kappa^{(l)}$



(d) $s^{(l)} = 3$, maximum $\kappa^{(l)}$

Figure 49: The encoder E and decoder D networks used for $t^{(l)} = 6$. Red arrows indicate the activation function σ_S, blue ones σ_T, green ones σ_R, and black ones σ_L (see equation (42)).